

AD-A158 102

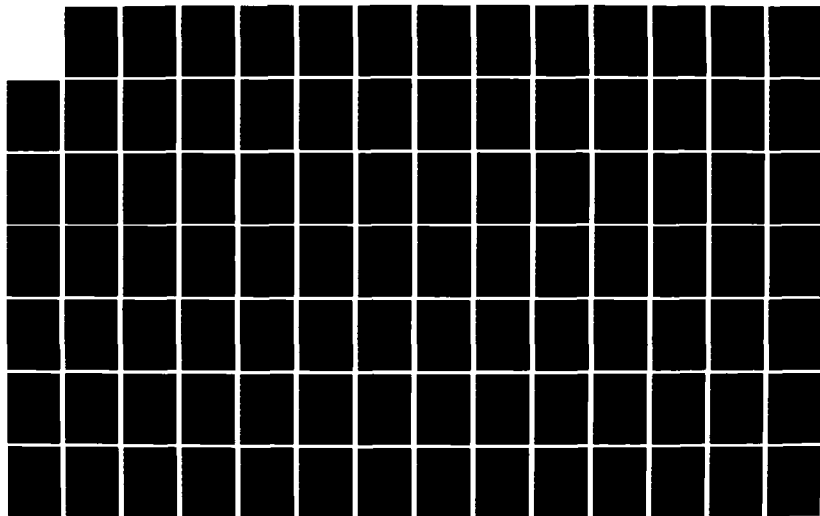
A META SYSTEM FOR GENERATING SOFTWARE ENGINEERING
ENVIRONMENTS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON
AFB OH W L MCKNIGHT 1985 AFIT/CI/NR-85-710

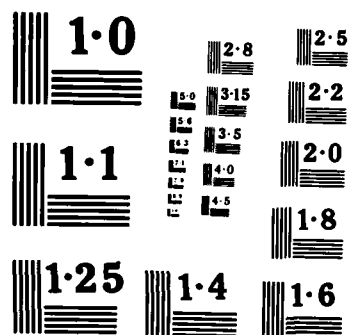
1/4

UNCLASSIFIED

F/G 9/2

NL





NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

AD-A158 102

1

**A META SYSTEM FOR GENERATING SOFTWARE
ENGINEERING ENVIRONMENTS**

BY

Walter Lee McKnight

Major, USAF

1985

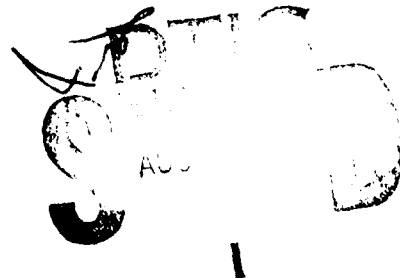
Pages: 277

Degree : Ph.D.

The Ohio State University

DTIC FILE COPY

This document has been approved
for public release and its
distribution is unlimited.



Copyright by
Walter Lee McKnight
1985

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 85-71D	2. GOVT ACCESSION NO. AD-4158 10	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Meta System For Generating Software Engineering Environments		5. TYPE OF REPORT & PERIOD COVERED THESTS/DISSERTATION
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Walter Lee McKnight		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: The Ohio State University		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433		12. REPORT DATE 1985
		13. NUMBER OF PAGES 277
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASS
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1 5 AUG 1985 LYNN E. WOLAVER Dean for Research and Professional Development AFIT, Wright-Patterson AFB OH		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) ATTACHED		



A-1

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

85 8 13 084

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to ascertain the value and/or contribution of research accomplished by students or faculty of the Air Force Institute of Technology (AU). It would be greatly appreciated if you would complete the following questionnaire and return it to:

AFIT/NR

Wright-Patterson AFB OH 45433

RESEARCH TITLE: A META System for Generating Software Engineering EnvironmentsAUTHOR: Walter Lee McKnight

RESEARCH ASSESSMENT QUESTIONS:

1. Did this research contribute to a current Air Force project?

☐ a. YES☐ b. NO

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not?

☐ a. YES☐ b. NO

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency achieved/received by virtue of AFIT performing the research. Can you estimate what this research would have cost if it had been accomplished under contract or if it had been done in-house in terms of manpower and/or dollars?

☐ a. MAN-YEARS _____☐ b. \$ _____

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3. above), what is your estimate of its significance?

☐ a. HIGHLY
SIGNIFICANT☐ b. SIGNIFICANT☐ c. SLIGHTLY
SIGNIFICANT☐ d. OF NO
SIGNIFICANCE

5. AFIT welcomes any further comments you may have on the above questions, or any additional details concerning the current application, future potential, or other value of this research. Please use the bottom part of this questionnaire for your statement(s).

NAME _____

GRADE _____

POSITION _____

ORGANIZATION _____

LOCATION _____

STATEMENT(s): _____

A META SYSTEM FOR GENERATING SOFTWARE
ENGINEERING ENVIRONMENTS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

BY

Walter Lee McKnight, B.S., M.S.

* * * * *

The Ohio State University

1985

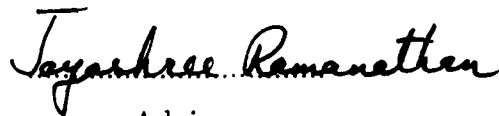
Reading Committee:

Dr. Jayashree Ramanathan

Dr. Sandra Mamrak

Dr. Venkataraman Ashok

Approved By

A handwritten signature in cursive script, reading "Jayashree Ramanathan".

Adviser

Department of Computer
and Information Science

A META SYSTEM FOR GENERATING SOFTWARE ENGINEERING ENVIRONMENTS

BY

Walter Lee McKnight, Ph.D.

The Ohio State University, 1985

Professor Jayashree Ramanathan, Advisor

TRIAD is a generic meta environment that has a knowledge base of methods from which a user can select a method to customize the TRIAD environment into a specific software environment. When a method is selected to customize the TRIAD environment, it becomes an instantiated method which provides guidelines and enforcement policies for developing and recording project information. The main contribution of this dissertation is in developing a kernel tool for TRIAD. This tool called the tuner can create method representations for the knowledge base of methods and can transform method representations already in the knowledge base of methods. The tuner is a parser generator like tool. However, instead of building representations in a batch and static mode, the tuner is able to create and transform the method representation in an incremental and dynamic way. Even when a method is instantiated and a part of the TRIAD environment, the method representation can be transformed. Methods are modeled by attributed grammar forms. A genesis grammar form was developed which can be used to generate all other grammar forms. The tuner is built upon the model of the genesis

grammar form. The tuner is able to reflect experience gained with using a method back into the method by program transformation on the method representation. The power of the tuner is illustrated by three examples discussed in the dissertation. Problems discovered while implementing the three examples that are associated with designing interfaces for software environment are also discussed.

Dedicated to my wife, Carol
and to my children Sam, Heidi, and Mary.

Acknowledgement

I would like to thank my advisor, Professor Jay Ramanathan, for her help and positive attitude during these past three years. If it wasn't for her faith and encouragement, I would never have completed this work in time. Many hours were spent in stimulating discussions that lead to this research topic. I am also grateful that she encouraged me to write down my ideas early.

I would also like to thank the other members of my reading committee, Professors Sandy Mamrak and Venkataraman Ashok, for their suggestion. Professor Mamrak helped me develop the ideas of canonical representation and their applicability to operating systems. Professor Ashok helped me with VLSI design and inspired me to look at how VLSI methods could be supported in TRIAD.

I wish to thank members of the TRIAD project, John Rose, Steve Demion, Merete Jordal, Ronnie Sarkar, Jim Kiper, Ed Michelini, Thorbjorn Anderson, Bill Hochstettler, Ron Hartung, and Greg Donnells. John is the artist and is responsible for the figures in this dissertation. Special thanks goes to those who first worked on the TRIAD project and who left something behind for the rest of us to work on.

I wish also to thank Dr. Chin Li work the many hours of discussion we had as we worked together in bringing up the second version of

TRIAD. It was his insights that lead some of the ideas of the tuner tool.

Last, but not least, I want to thank my dear wife, Carol, for her understanding and support. She willingly let me stay behind and work instead of taking her and the kids on vacation. She took on a bigger load of raising our three children, Sam, Heidi, and Mary, so that I would have more time to do this research.

Vita

EDUCATION and EXPERIENCE

September 28, 1947	Born, St. Ignatius, Montana
June 1972	B. S. Mathematics, University of Utah Salt Lake City, Utah
December 1973	M. S. Computer Science, University of Utah, Salt Lake City, Utah
1974 - 1978	System Analyst, NORAD, USAF, Colorado Springs, Colorado
1978 - 1982	System Analyst\Programmer, NATO, USAF, Glons, Belgium
1981 -1982	Instructor, Department of Mathematics, City College of Chicago, Glons, Belgium
1982 - present	Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

Research Interests

1. Integrated Software Environments: customizability and extensibility of environments, integration of software and project information, method support for all phases of a project, automation of method support using techniques synthesized from compiler technology, database management systems, and operating systems.
2. Software Tool Design: integrating existing software tools into a more useful, human-engineering environment, and development of tools which allow engineering knowledge into environments.
3. Specification of Data Abstractions: specification techniques, communication in terms of abstract objects, their automatic/semi-automatic implementation and high level debugging.
4. Other interests: operating systems, programming language theory, compiler theory, and software engineering.

- Tool Integration: Software engineers need to make a uniform interface through which tools can access method structured information in the project information database. This interface must be general enough to allow easy addition of new tools as they become available.
- Method Support: Although software engineers should not impose any particular method in the design of a software programming environment, they should build support for the methods practiced by the intended users of the system. Support should involve explicitly displaying method information such as module interface specifications, instructions for tool use or formats for various representations of the project information. This would result in greater standardization of software system produced and reduce the time to educate new members in the company's design and coding practices.

To get a further perspective (see figure 1) on the degree of support that is possible for methods, it is useful to get another global view of the software engineering tasks. The problem of engineering a software product is complex because of two fundamental types of tasks [4, 5]:

- Information handling tasks related to organizing large volumes of information, composed of functional specification, design documents, interface specifications, etc., for the numerous subsystems generated during the design process, and
- Complex problem solving tasks which consist of domain-oriented decisions about the content of the abstractions used

In this dissertation, we will look at how the TRIAD system has addressed these problems of describing methods, of maintaining a knowledge base of methods, and of customizing an environment. We will see how the model we developed for describing methods allows us to make changes to a method and to reflect these changes into the project information database. We will also show how the TRIAD system can easily be customized to any method.

1.2. General Requirements for Improvements

Based on a wide variety of industrial case studies and a detailed look at examples such as the one discussed in the previous section, we have observed that the deep differences in project types among various industrial settings have contributed to the evolution of numerous methods for the software engineering process. Fundamentally, a software engineering method organizes and manipulates software-related information (requirements, specifications, management plans, design, code, etc.). The role of a software engineering environment is to provide support to view this information, manipulate it, and apply tools to it [33].

Software engineers creating software/programming environments are faced with three requirements:

- Logical organization of total project information: Methods should suggest the logical organization of the project information. This information should be chunked so that meaningful queries can be asked. Software engineers should use domain specific information about specific methods to optimize the design of the project information.

the type by using an external signal port, a component can be made externally available by defining it to be an externally-accessible component. This eases a restriction of types in a controlled way.

Suppose a company had built a VLSI design environment with a hard-wired Mead-Conway type-based method [73]. Now suppose that the company wanted to design printed circuit boards which are best designed using a flat method. This company has the following options for designing printed circuit boards:

1. It could develop a new VLSI design environment for a flat method to be used in designing printed circuit boards.
2. It could develop a new VLSI design environment for the Sheet/Type Method and design of all its products using the new environment.
3. It could force the printed circuit boards to be developed with the type-based method that is already hard wired in the current VLSI design environment.

None of these solutions are acceptable. Both solutions 1 and 2 cost time and money which makes the designing of the printed circuit boards neither profitable nor competitive. Solution 3 makes the design of the printed circuit board both cumbersome and inefficient. Hard wiring a method limits flexibility when dealing with new problems. If the VLSI design environment has a knowledge base of methods, then adding a method to handle the new design of printed circuit boards would be as simple as adding a new method to the knowledge base.

is partitioned into a multi-level hierarchy of modules, each of which must be an instance of a type. A type is a black box. All internal structures are encapsulated in the type and the only access is through a well-defined interface. This definition of a type provides high security. As long as the designer does not change the interface or functionality, any internal change desired can be made without worrying about whether the change will adversely affect anyone else using the type.

Each type communicates only through a well-defined interface; communication using global signal names is severely restricted. It is not possible to describe overlapping structures. Type-based design is well suited to a simple cell-based VLSI design. The instantiation capability missing from flat design is exactly what is needed for regular arrays of cells. However, the lack of global signals and sharable components inhibits some forms of design.

One approach to overcome the disparity between the flat and the type-based design methods is the Sheet/Type Method introduced by IBM [6]. The two concepts of this method are:

1. A module can be either a sheet or a type. If it is a type, it communicates only through a well-defined interface and its components cannot generally be shared. If it is a sheet, it communicates both through an interface and by using global signal names. Its components can be shared by other sheets. Types and sheets have components which can be types or sheets, in any combination.
2. The interface concept of a type is extended to include components. Just as a signal can be made available outside

1.1. An Example: Current Problems in Design Methods Implementation

The general problems associated with method implementation are illustrated with an example from the VLSI design process domain. In no other place has the proliferation of methods had a greater impact than in the VLSI design process domain. This proliferation has been caused because of the many different technologies and the many interactions between cells. With each new technology, a new and better design system has been developed to implement that technology and its associated method. Let us illustrate this with an example.

There are two existing classes of methods for VLSI design; flat design and type-based design. Using the flat class, the entire design is viewed as being a collection of sheets all at the same level. The sheets communicate with each other by using global signal names. An additional feature of flat design is that a component shown in one sheet can also appear in other sheets (i.e. the connections to a component do not have to be shown on one sheet). This feature is used frequently in printed circuit board designs where the power and ground connections are shown separately from the logic flow.

The main shortcoming of flat design is that it has no special provision for exploiting regularity in a design (i.e. where a certain circuit topology is used more than once). A sheet cannot have multiple instantiations. Although it is possible to make copies of sheets, there is no mechanism for ensuring that these copies remain consistent throughout the life of the design.

Type-based design follows a completely different approach. The design

Each company would develop their own software environment system which would enforce their methods and policies. If, after some experience with the method and the management policies, the company decided to change the format and placement of the comments, they had a tremendous task of updating the software environment system to reflect these changes.

In this dissertation, we will look at how to implement methods in a software environment system. We will develop a model in which methods can be added to the system, in which methods can be developed for the system, and in which methods can be tuned for the system. In essence, we will develop a meta environment system which has a knowledge base of methods. From this knowledge base, a method designed for a particular project can be selected and used to customize the meta environment. As users of the system use this method, their experience can be obtained and tuned back into the method itself and reflected in the project information database as well. A software environment system does not have to be hard wired to any particular method.

In section 1.1 we will look at the proliferation of methods and its effect on design systems. In section 1.2 we will look at some of the tasks that need to be performed by methods and how they could be incorporated into design systems. In section 1.3 we will look at the contributions I have made to computer science as they apply to methods and meta environment systems. We will also see how the rest of the dissertation is organized and how it addresses the issues of methods and meta environment systems.

3. a set of predictable milestones that can be reviewed at regular intervals throughout the project life cycle and used to evaluate the progress of the project.

Although the objectives were well understood, the implementation of a method on a software environment system was both cumbersome and irrelevant. Software produced by these systems became more expensive to produce, less responsive to user demand, took longer to implement, and was harder to maintain. One reasons for this was that the software environment system was so tied to a particular implementation of a method that any change in the method meant major changes to the system. The experience gained from the use a method could not be reflected in the software environment system without affecting every project.

Many management policies, which are not applicable from one company to another, were placed in the methods and supported by the software environment system. For example, the project life cycle method says that all code should be documented. Each company would have its own policy on how the code was to be documented. One company might require that comments be intermixed with the code. Its software environment system then would enforce the policy that code would be found in the first 40 columns, and the comments would be found in the last 40 columns. Another company might have its system enforce the policy that all comments be placed at the beginning of a procedure or module and that all comments include such details as which routines are called, what are its inputs, what are its outputs, and what side effects are present. Still another company might have its system enforce the policy that all comments be in the form of a flowchart.

As the cost of hardware became less expensive, more individuals began to see how computers could make their work easier. These individuals were not programmers, nor did they understand how computers worked. They wanted the computer to do more complex tasks for which algorithms were not clearly defined. They also wanted documentation to show them how to use the product they had asked for. Because the programmer who was responsible for maintaining the code was not the same one who originally designed the code, requirement documents, functional specification documents, and design documents became necessary. Programmers had to communicate with one another as well as with the users. Still, there were not any methods to help the programmer in designing, coding, and testing of programs. Management tried to set up some policies to manage the programmer and their products, but these policies were often ill-placed or not well understood.

In the early 1970's, a new discipline came into existence known as software engineering. It was based on a model of engineering that had been used to manage hardware development. Several methods were developed by individuals like Jackson [49], Yourdon [110], and Constantine on what they considered to be good programming practices. These practices were combined with what management considered to be good ways of managing a programming project and formed project life cycle methods. The objectives of these project life cycle methods were:

1. a well-defined method that addresses a project life cycle of planning, development, and maintenance,
2. an established set of software components that documents each step in the life cycle and shows traceability from step to step, and

Chapter 1

Introduction

"There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success, than to take the lead in the introduction of a new order of things..."
(Machiavelli)

Computer programmers have suffered from the perils of building a system that does not work like the user wants it to, that was not installed when the user wanted it, and that does not lend itself to meet the future needs of the user. They in essence suffer the Machiavelli syndrome mentioned above.

When computer programmers first began programming in the early 1950's, no methods were used in their programming. They wrote code that was full of tricks to make it run efficiently, but so obscure that no other programmer could work on it. Documentation consisted only of a very simple user's manual, generally done as an afterthought. Although the code may not have worked like the user wanted, at least it was installed in time and it did not cost as much as the hardware on which it ran. The problems that were being programmed were based on well known algorithms and were basically numeric in nature. In a lot of cases, the users were the programmers themselves (or at least individuals who understood programming).

Figure 31:	Algorithm for Intersection Set	164
Figure 32:	Example of Common List Filled Form	165
Figure 33:	Example of Noncommon List Filled Form	166

List of Figures

Figure 1:	Categorization of Tasks in a Software Engineering Process	11
Figure 2:	TRIAD vs a Data Base	23
Figure 3:	Concept Tree for a Book Grammar	24
Figure 4:	Interpretation Tree for a Book Grammar	25
Figure 5:	Blank forms for the Book Method	26
Figure 6:	An Instantiation of the Book Form	27
Figure 7:	Instantiation of the Chapter and Section Forms	29
Figure 8:	Comparison of Programming Environments	32
Figure 9:	Methods For the Project Life Cycle	47
Figure 10:	Relationship Between a concept tree and its interpreted concept tree	52
Figure 11:	Derivation of $((()))$	63
Figure 12:	Derivation trees of $x \times y + x$	66
Figure 13:	Derivation Tree of 673	80
Figure 14:	Dependency Graph for Machine-Dependent Pascal	82
Figure 15:	Methodology Form of the Meta Methodology	114
Figure 16:	Grammar Form of the Meta Methodology	115
Figure 17:	Action Form, Attribute Form, and Symbol Form of the Meta Methodology	116
Figure 18:	Production Form of the Meta Methodology	117
Figure 19:	An Instantiated Methodology Form	118
Figure 20:	Concept Tree for a Blank Methodology Form	124
Figure 21:	Concept Tree for an Instantiated Methodology Form	125
Figure 22:	Locally Tuned Form	129
Figure 23:	An Updated Fillform of the Grammar Form	143
Figure 24:	Verify Name Not in Table Algorithm	144
Figure 25:	Verify Name is in Table Algorithm	145
Figure 26:	Diagram of VLSI Method	148
Figure 27:	PEG Blankform of the VLSI Method	152
Figure 28:	EQNTOTT Blankform of the VLSI Method	153
Figure 29:	Algorithm for Tool Interface	155
Figure 30:	Define Domain Forms for Virtual Interface	162

4.6.1. Normal Form	104
4.7. Attributed Grammar Forms	110
5. TRIAD and the Grammar Form Model	112
5.1. Description of a Method in TRIAD	113
5.2. Tuning a Method	125
5.2.1. Tree Rebinding at Method Use Time Tuning Example	127
5.2.2. Tuning Example of Concept Rebinding at Method Use Time	128
5.3. Why Attributed Grammar Form Model	132
6. The Tuner	135
6.1. Description of the Meta Method	136
6.2. Implementation Issues of the Meta Method	138
6.2.1. Implementation Issues of Procedural Components	139
6.2.2. Implementation Issues of Attributes	142
7. Using the Tuner for a VLSI Method	147
7.1. Description of the VLSI Method	147
7.2. Implementation Issues of the VLSI Method	151
8. Using the Tuner for a Virtual Interface Method	156
8.1. Description of the Virtual Interface Method	156
8.2. Implementation Issues of the Virtual Interface Method	160
9. Conclusions and Future Work	167
9.1. Common Properties of Methods	169
9.2. Multiple Display Interfaces	170
9.3. Data Structures For Attributes	171
9.4. Compiler For Procedural Components	173
9.5. Data Base Model	174
9.6. Generic Procedural Components	175
9.7. More General Attributed Grammar Form	176
Bibliography	178
Appendix A. Implementation of the Tuner in Tuner	189
Appendix B. Implementation of a VLSI Method	216
Appendix C. Implementation of the Virtual Interface Method	252

Table of Contents

Acknowledgement	iii
Vita	v
Table of Contents	viii
List of Figures	x
1. Introduction	1
1.1. An Example: Current Problems in Design Methods Implementation	5
1.2. General Requirements for Improvements	8
1.3. Contributions	10
1.3.1. Development of a Model for Describing Methods	12
1.3.2. Development of a Tuner Tool for the Incremental Design of Methods	12
1.3.3. Method Transformations	14
1.3.4. Problems With Design of Interface for Software Environments	15
1.3.5. Validation of New Ideas With Realistic Examples	16
2. Current Research	18
2.1. A Brief Introduction to TRIAD	21
2.1.1. Implementation View of TRIAD	22
2.1.2. User's View of TRIAD	24
2.1.3. Architecture of the TRIAD Meta System	28
2.2. Comparison with Other Systems	31
3. Methods and Methodology	44
3.1. What is a Method?	45
3.2. Steps in the Evolution of Methods and Their Tools	48
4. Attributed Grammar Forms	53
4.1. Introduction	53
4.2. Context Free Grammars	55
4.3. Generative Grammars	61
4.4. Attributed Grammars	73
4.5. Attributed Grammars with Right Regular Parts	84
4.6. Grammar Forms	88

Publications

1. "A Meta System For Generating Software Engineering Environments", submitted to Second Conference on Software Development Tools, Techniques, and Alternatives, San Francisco, California, December 1985, co-author: J. Ramanathan.
2. "Opportunities and Approaches for Using Artificial Intelligence Techniques in Practical Software Engineering Environments," Technical Report TRIAD-7, August 1984. Co-authors: J. Ramanathan, S. Demion and C. Li.
3. "Uniform Support for Information Handling and Problem Solving Required by the VLSI Design Process", ACM IEEE 21st Design Automation Conference, Albuquerque, New Mexico, June 1984, co-author: V. Ashok and J. Ramanathan.
4. "Integrated Environments For Information Management in VLSI Design", Proceedings of the Computer Data Engineering Conference, Los Angeles, California, April 1984, Co-author: V. Ashok and J. Ramanathan.
5. "A Generalized Record Parser", Master thesis, Department of Computer Science, University of Utah, December 1973.

to describe the subsystems and the steps needed to eventually transform the various abstractions into code.

These two types of tasks are related (see Figure 1) because for problem solving to be well-informed, appropriate requests for views of information must be formatted and pertinent information views must be displayed to the user. For example, to make a well-considered decision regarding the content of a subsystem, the designer must be presented with a view of the information which documents the functionality of all the related subsystems.

In this dissertation, we will see how the method support can be made apart of the environment. We will also see how the logical organization of project information is supported in TRIAD and how this supports complex problem solving.

1.3. Contributions

The contributions of this dissertation are in:

- developing a model in which methods can be described,
- developing a tuner tool for the incremental design of any method,
- expanding the ideas of method transformations,
- solving the problems associated with the design of the interface for software environments, and
- validating each of the above by using realistic examples from three different domains.

The contributions are discussed below.

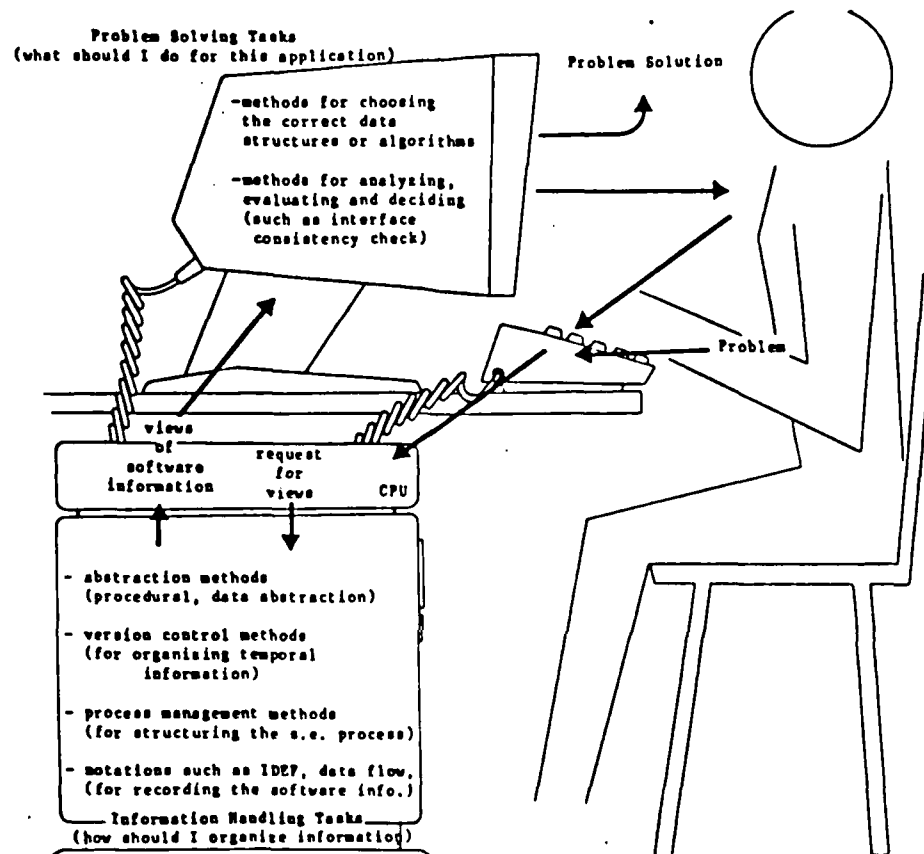


Figure 1: Categorization of Tasks in a Software Engineering Process

1.3.1. Development of a Model for Describing Methods

Early software engineering methods were characterized by a step wise refinement process. Later, methods became more complex requiring back-tracking, recursion, alternatives, etc. For example, many methods have an enforcement policy that requires certain steps to be accomplished before other steps. An attributed grammar form model is introduced which allows a method designer to describe a method where

1. enforcement policies can be handled by using attributes and procedural components,
2. automation of certain steps can be handled by including tools which can be interfaced with the system by the use of global controllers and procedural components, and
3. back-tracking, recursion, alternatives can be handled by a context-free grammar.

The evolution of methods and how they can be described is discussed in chapter 3. Grammar forms are discussed in chapter 4. How a method can be described by a grammar form is discussed in chapter 5.

1.3.2. Development of a Tuner Tool for the Incremental Design of Methods

Grammar forms themselves represent a family of grammars. I have defined a genesis grammar form which is powerful enough to describe the form grammar and the allowable transformations needed to define any grammar form. With this genesis grammar form, I can describe any grammar.

The prototype TRIAD (Tree based Information Analyzer and Developer) is an integrated, tool-box enriched, method driven environment designed to support the entire spectrum of software engineering activities rather than simply programming activities. One of the tools in its tool-box is the tuner tool. This tool creates method representation to be included in the knowledge base of methods that TRIAD uses to customize itself into a practical software environment.

I developed the tuner tool that uses the genesis grammar form. In the previous version of TRIAD, all methods had to be hand coded and compiled with TRIAD. It was difficult to implement new methods because the method designer had to understand the code of TRIAD to bring up a new method.

The first version of the tuner tool just eliminated this hand coding of methods. It made the method independent of TRIAD code. This version also introduced the grammar form model into TRIAD. The second version of the tuner allowed attributes to be defined and semantic routines to be executed. The semantic routines had to be hand coded with TRIAD because there was no interpreter for them. The next version included an interpreter for semantic routines which were expanded to procedural components. Each of the versions of the tuner were static versions working off of the genesis grammar form.

The last version of the tuner allowed for making dynamic changes to a grammar form once it had been instantiated. This version allowed a method to be incrementally built, dynamically built, and dynamically changed and to have those changes reflected in the refinement tree of the product being developed. This tuner allowed the user more freedom to experiment with the development of a method. It also allowed the user

to reflect experience back into the method itself. Several individuals have used this last version of the tuner and have found it easier to use than any of the previous versions.

The genesis grammar form is defined in chapter 3. How this grammar form is used in TRIAD is explained in chapter 5. Also included in chapter 5 is how the tuner tool uses the grammar form model, how the tuner tool can make changes to a method and have it reflected in the refinement tree of the product.

1.3.3. Method Transformations

In building the tuner tool, extensions to the parser generator concept were developed and tested out. The tuner is comparable to a parser generator in that it builds tables to drive a generic system. A comparison of the tuner and a parser generator follows:

- The tuner has to understand the model used to represent methods. This model is an attributed grammar form. A parser generator also has to understand the model used to represent programming languages. This model is often an attributed grammar.
- The tuner has to manipulate the method representation and the method instantiation and keep them both consistent. There is no way that a parser generator can manipulate the language representation and have it reflected in the programs using that language. For one thing the language representation is not maintained in the program. If the language representation is changed by the parser generator, a

new compiler has to be generated and the program run through the new compiler. This is a three step operation compared to the one step operation of the tuner.

- The tuner has to work in an incremental and dynamic mode. This allows the user of the method to make changes which are appropriate to the project being developed and to see the results immediately. To have the same concept work with a parser generator would mean that we could tune a programming language to the particular program being developed. In essence we could change features of a programming language to optimize the problem being solved. Instead of having a hundred variations of a programming language, each emphasizing a particular concept of programming language theory along with a compiler for each of these languages, we could have a few classes of languages and a compiler for each class of languages. Then the user could tune the programming language to reflect the experience accumulated and this could be propagated to the compiler without the user having to write a new compiler.

1.3.4. Problems With Design of Interface for Software Environments

Any method is only as powerful as its presentation to the user. Because of the large amount of project information that must be captured in a software environment, only a small portion of the total project information can be presented to the user at any given time. This has lead to several problems with the design of software environments. Among these problems are the following:

1. How to display the project information without showing its hierarchical nature.
2. How to engineer the grammar form production without having to take into consideration the display interface with the user.
3. How to engineer a display interface that can develop the underlying refinement tree without actually displaying the underlying refinement tree.

These issues are addressed in the three examples found in chapters 6, 7, and 8.

1.3.5. Validation of New Ideas With Realistic Examples

As a software engineer, I believe that ideas are only as good as they are useful. One of the real problems in computer science is showing that your ideas have a practical application. I have chosen three areas to show where the tuner tool could be used in practical application.

The first area was in the software engineering domain. The application that I thought would best illustrate the power of the tuner tool was to dynamically build the meta method using the tuner tool. The tuner tool understands the meta method, so in essence I built the meta method using the meta method. Since the tuner in many respects represents a parser generator, this example show how the ideas of the tuner could be expanded to compiler theory. The example is explained in detail in chapter 6.

The second area was in the VLSI design domain. The example for

this domain was the development of a method to design finite state machines using generic tools like a state diagram generator, a truth table generator, PLA generators, simulators, and formatters. This method helps the student to be more aware of what tools are available and takes care of some of the automatic tasks associated with using the tools. This example is detailed in chapter 7.

The last area was in interface design. Another student has designed a method to build a virtual interface. In this method there were several steps that could be automated. In conjunction with this student, we used the tuner tool to implement this method and automated several of the steps to show its feasibility. The details of this example are found in chapter 8.

Chapter 2

Current Research

Research in programming environments began in the mid 1970's. Research in software environments began in the late 1970's with the initial work on the TRIAD project beginning in 1979. Since that time many programmer oriented systems have been designed and implemented. These systems range from program/structure editors systems and office automation systems to programming environments and software environments.

TRIAD is unique in that it is can be adapted to any or all phases of a project life cycle. Its knowledge base of methods customizes the system to support the structuring and manipulating of all software-related information generated during the entire project life cycle.

In this chapter we will take a look at the TRIAD system both from an implementation point of view and from a user's point of view. We will then compare the TRIAD system with other programmer oriented systems with emphasis on the contributions TRIAD has made to software environments.

We will basically ignore office automation systems like SOFTFORM [48] and the one proposed by Shu [90] because their emphasis is more on office automation for a non-programming environment than support for a programming environment. Most office automation systems have a

forms base like TRIAD, but do not have a grammar or attributed grammar model supporting them. We will also ignore interface systems like the Box [16] and Cousin-Spice [44] because those systems address the problems associated with building programming environments more than being programming environments. Because of the number of other systems, we will only look at a few. The list of systems that we will compare with TRIAD includes:

1. ALOE -- CMU [30, 31, 74]
2. CPS -- Cornell University [84, 85, 102, 103]
3. SUPPORT -- University of Maryland [111]
4. POE -- University of Wisconsin [32]
5. COPE -- Cornell University [3]
6. SYNED -- Bell Laboratory [46]
7. PECAN -- Brown University [82, 83]
8. MAGPIE -- Tektronix [19, 89]
9. ARCTURUS -- University of California [97, 98]
10. SAGA -- University of Illinois [11, 12]
11. ISDE -- System Management, Italy [14]
12. MENTOR -- INRIA, France [26]

13. GANDALF -- CMU [36, 41, 42, 43]

These systems cover a range of support activities that should be found in a software environment. The list includes program editor systems (MENTOR, SYNED), a program transformation system (SUPPORT), a program editor generator (ALOE), program development systems (CPS, PECAN, POE, COPE, MAGPIE), and project development systems (GANDALF, ARCTURUS, SAGA, ISDE). The comparison will be similar to the comparison made by Kuo in [61] although our emphasis will be more on how they support a multiplicity of method driven activities as opposed to what they can do. We realize from the beginning that they did not have the same design goals that TRIAD had and so we will not attempt to evaluate the other systems with each other.

TRIAD differs fundamentally from other programming environments because it has a knowledge base of methods (like Jackson [49], SADT [87, 88], etc). Basically, a method is a sequence of steps that focus on the activity at hand. A method could be very general and could support, for example, version control and other development-in-the-large activities. A method could also be very domain specific and could support coding and other programming-in-the-small activities. The more useful methods might be a synthesis of other methods and could support both in-the-large and in-the-small activities or even design/development activities in other engineering disciplines. A method representation can be user-selected to customize the TRIAD interface. This representation customizes the software environment for the user. A detailed discussion of what is a method and how it has evolve can be found in chapter 3.

TRIAD's knowledge base concept has provided some unique

opportunities for research into software productivity issues. Some of the major ones are:

- What are the mechanisms which constitute a meta environment?
- How can one represent a method to utilize these mechanisms?
- What is a good method for a specific application domain?
- What are some relatively intelligent ways to use project information developed using a method?
- How can one tune an existing method to reflect experience in an application domain?

These issues will be addressed throughout this dissertation.

2.1. A Brief Introduction to TRIAD

TRIAD is a form based environment developed as a meta environment for software development [81]. Currently, there are two prototype versions, one on a VAX/11780 running UNIX and the other on a IBM/4341 running VMS. In section 2.1.1, we will discuss how TRIAD looks from a implementation point of view in order to motivate some of the deeper architectural issues underlying TRIAD. Then in section 2.1.2, we will discuss how the user views TRIAD. Finally, in section 2.1.3, we will look at what is needed in order to architect a meta environment.

2.1.1. Implementation View of TRIAD

The model underlying TRIAD is an attributed grammar form [72]. We will give only a brief explanation of the model here for comparison purposes, the formal details are explained in chapter 5.

Most of the other systems use an attributed grammar model which is inadequate for modeling a meta system for the entire software engineering process. An attributed grammar model does not allow the dynamic customization and tuning of the meta environment. Those systems which use a grammar base model are even more limited in that all they can emphasize is the coding process.

A grammar form has a concept grammar and an infinite vocabulary. The concept grammar generates concept trees which are hierarchically related layers of tags on top of the data base (see figure 2). The infinite vocabulary is the chunk of text associated with each tag in the concept tree. Symbols of the concept grammar can be substituted with a subset of symbols from the infinite vocabulary to derive various interpretation trees. In fact, this process of substitution is called interpretation. As we shall see later, the tuner tool exploits the interpretation process.

Conceptually, the concept grammar is analogous to the data definition language for a hierarchical database. Individual concept grammar symbols correspond to individual entities and concept grammar productions rules correspond to relationships between entities. An instance of an entity is a chunk of project information. In addition, each tag in the concept tree serves to structure chunks of project information according to the model on which the concept tree was

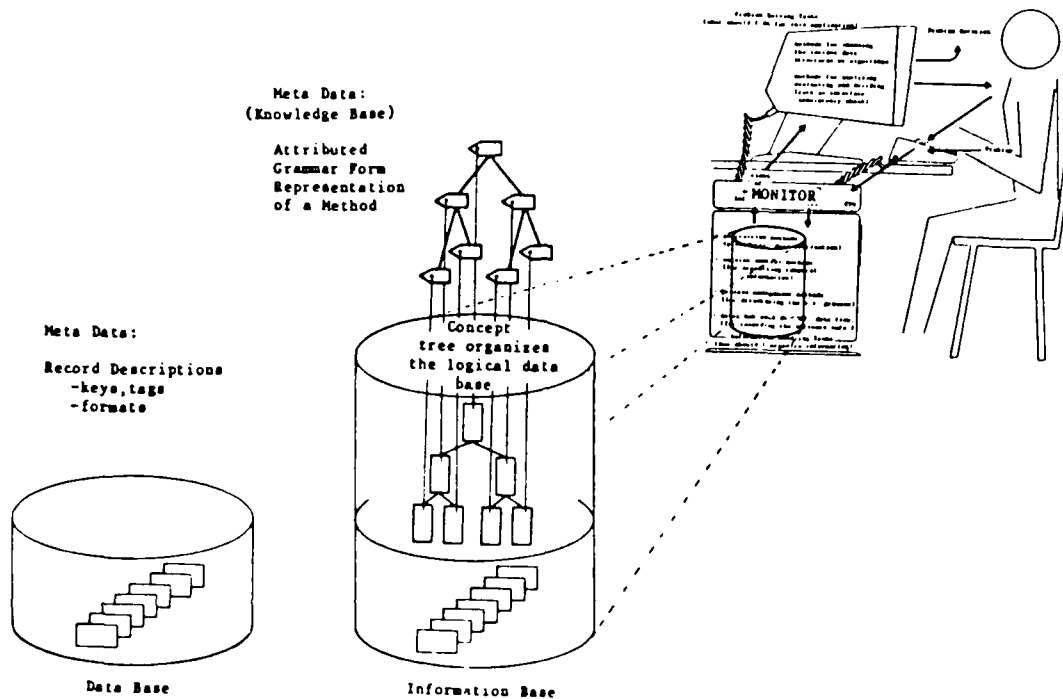


Figure 2: TRIAD vs a Data Base

designed (see figure 2). In a data definition language, there is only one definition of an entity, which does not reflect the model upon which it was built. In TRIAD, we are able to capture this model with the concept grammar.

Let us illustrate the system view with an example. Let us assume we have a method for describing a book. The concept grammar production rules include:

1. (book) ::= (title, author, *(chapter)*)

These descriptions are used to produce tables and code to direct the language-dependent modules of the system. The language description for PECAN is given in four parts. The principle part describes the abstract syntax trees which includes the semantics of each abstract syntax production as well as information to control the parser and editor. The remaining parts are more detailed semantic information about the use of symbols, data types, and expressions in the language. This generator works only in a static mode like the CPS generator.

POE is a full-screen language-based editor that knows the syntactic and semantic rules of Pascal. POE is structured more like CPS, but its philosophy is more like COPE in that it does not use templates. It presents an interface in which the user moves the cursor to a prompt symbol and types text corresponding to the prompt. Typing a single-token prefix of a particular expansion is sufficient; an automatic syntactic error corrector provides any added tokens which are necessary to expand the user's input and make it syntactically correct. There is an editor-generating system called POEGEN which creates the language-specific characteristic of POE in a table format similar to the CPS generator. POE represents an open environment. This means POE can read text files created by any program or tool and can output text files usable by other programs or tools.

Integrated programming environments are usually described as a set of tools that support program creation, modification, execution and debugging. The term integrated emphasizes the difference between the programming environment and loosely coupled sets of tools that most programmers typically use: an editor, compiler, assembler, linker, loader, and debugger. The tools in an integrated programming environment

The COPE system is similar to CPS except it is based on an intelligent parser. The parser is able to insert missing keywords and tokens to get about the same affect as the templates do in the CPS. This scheme has the advantage that the user can type in their program as text. It has the disadvantage that the user is not shown what templates are currently valid. The primary focus of the system is for automatic error recovery and repair. COPE has an UNDO and a REDO facility that allows for experimentation of coding. No additional tools are planned to be included in the COPE system. It is based completely on the PL/CS language and has no method associated with it. There is no generator for the COPE editor.

The PECAN system differs from the other program development systems in its use of multiple views of the shared data structures. A program is represented internally as an abstract syntax tree. The user does not see this tree directly, but instead sees views or concrete representations of it. Some of the views include the syntax-directed editor view, the Nassi-Shneiderman structured flowchart view, and the module interconnection diagram view. PECAN supports both the template-based approach of CPS and the text-based approach of COPE. PECAN is designed to be interactive and to eventually support graphical programming similar to what is done in the Smalltalk system [40, 94], the Interlisp environment [104], the Mesa environment [37], and more recently, the CEDAR environment [23]. PECAN differs in its emphasis on programming-in-the-small, interaction, the use of graphics, and on showing the user multiple views of his program.

A PECAN program development system is generated for a particular language from descriptions of the syntax and semantics of the language.

modification, re-evaluates the attributes, and detects if there are data-flow anomalies.

It is from the CPS Generator that the original ideas and models for the TRIAD tuner were developed. Since that time the tuner has been modified to the point where there are major differences between the generator and the tuner. These differences include:

- The generator is based on a model of attributed grammar where the tuner is based on a model of attributed grammar forms. With the attributed grammar forms, the tuner can build editors for families of grammars where the generator can only build editors for a single grammar.
- The generator only allows the creation of semantic functions which means that information can propagate only up and down the tree. The tuner allows the creation of action routines which allow information to propagate across the tree as well as up and down the tree.
- The generator only works in a static mode. The generator builds the appropriate tables and those tables are used by CPS. The generator cannot work with CPS directly to update the tables while CPS is using them. This is not true with the tuner. The tuner can work either in a static or a dynamic mode. In the static mode it builds the tables for TRIAD. In the dynamic mode, it works with TRIAD and updates the tables the same time that TRIAD is using the tables. The tuner knows the consistency issues and is able to handle them.

design language built into the system which is algorithmic in nature. This allows the programmer to input the design of the system in an almost English like fashion and build layers of refinement of that design until the actual code is generated. The expansion is always closer to Pascal source code. The user has the option of displaying the design, the source code, or both intermixed. Actually the design becomes the documentation for the next lower level of refinement. Like SYNED, SUPPORT maintains both an abstract syntax tree representation and a text representation. SUPPORT allows both a top-down refinement and a bottom-up development of a system. When SUPPORT gets to the source code, it is interpreted and executed as it is inputted so that some semantic errors can be flagged. Because SUPPORT is a program transformation system, it is impossible for it to support more than one language at a time. All transformations are programmed into SUPPORT and are not input via some syntax tree. The system likewise can not be built dynamically nor incrementally. All changes to the system would cause a new source code to be generated for each design. All tools used by the system must be part of the system.

The Cornell Program Synthesizer (CPS), a program development system for PL/C, includes a syntax-directed editor and an interpreter. Its editor is template-based but provides text editing for fixed constructs like expressions. It has been implemented as a generator so that it is possible to create synthesizers for different block structured languages using attributed grammars to describe the display format and semantics for each production of the abstract syntax. All tools that interface with CPS work off of the abstract syntax tree and must be included in CPS. At present these tools work in an incremental mode. When a node of the abstract syntax tree is modified, the interpreter interprets the

All tools that interact with the MENTOR system must understand and use the abstract syntax tree.

SYNED is a grammar based program editor for the C language programmer. It manipulates a program as if it were text; however, it represents a program as an abstract syntax tree. The abstract syntax tree can be unparsed back into text. Both text and abstract syntax tree representations are maintained. Because of this desire to maintain two representations of the program, the user can either update the text representation and have it parsed into the abstract syntax tree representation or the user can update the abstract syntax tree representation and have it unparsed into the text representation. Documentation is maintained as comments attached to certain nodes in the abstract syntax tree. SYNED only works for the C language and therefore we can consider the system as neither adaptable nor customizable. No method is directly supported by SYNED, although with the way abstract syntax trees are built, top-down refinement could be considered the inherent method. One of the unique features of the SYNED system is the capability to undo a command, which allows for greater experimentation than with the MENTOR system. This feature is not currently supported in the TRIAD system. All interfacing with other tools is with the text representation, which is the common representation that most tools have expected in the past.

SUPPORT is a program transformation system. It is concerned with both the initial design, detailed design, and coding phases of a project and with programming-in-the-small issues. The initial design must be on a module by module basis. The inter relationships between modules can not be a part of the designing except as comments. There is a program

	Grammar based	Attributed-grammar based	Knowledge base of Methods	Multi-language	Use time binding	Tools internal	Tools external	Programming-in-the-small	Programming-in-the-large	Preliminary phase supported	Detail design phase supported	Coding phase supported	Testing phase supported
GANDALF/ALOE			X	X		X		X		X	X		X
CPS			X	X		X		X		X			X
SUPPORT								X		X		X	X
POE			X	X		X			X	X			X
COPE		X								X			X
SYNED		X							X	X			X
PECAN			X	X		X		X		X			X
MAGPIE			X					X		X			X
ARCTURUS			X					X		X	X	X	X
SAGA			X	X		X		X		X	X	X	X
ISDE			X	X		X		X		X	X	X	X
MENTOR		X						X		X			X
TRIAD			X	X	X	X	X		X	X	X	X	X

Figure 8: Comparison of Programming Environments

MENTOR is a grammar based system for the Pascal language programmer. Its data is only represented as an abstract syntax tree. The MENTOR system is built using syntax tables of Pascal. Documentation is maintained as comments attached to certain nodes in the abstract syntax tree. Although new procedures can be added to a MENTOR system, the system itself cannot be built dynamically nor incrementally. The main function of MENTOR is to make program transformations from the input program to the final compiled program.

1. Development-in-the-large [22]. These activities have to do with defining the architecture of the product in terms of its components, interfaces of the component modules, tracking versions of code, integration testing, etc. The enormous volume of product information (composed of both code and documentation) makes these activities very complicated.
2. Programming-in-the-small. These activities have to do with designing individual product components, using algorithms, selecting data structures, coding in a language, etc. These activities have been studied quite extensively and are well understood.
3. Project management. These activities have to do with the process of manufacturing and assembling the product components. Specifically, cost estimation, quality control (walk-throughs, inspections, etc.) and process monitoring play a large role. The more complex a product becomes, the more complex the process.

2.2. Comparison with Other Systems

In this section we will compare some of the other programming environment systems with TRIAD. A summary comparison of these systems is shown in figure 8. Notice, that only TRIAD has a knowledge base of methods and binding of the method or grammar to the actual system at execution or use time. The other systems have the grammar definition entered at compile time, not execution time. The knowledge base of methods and the execution time binding are two major distinctions of the TRIAD system.

- add new attributes and procedural components for extracting views for tools (i.e. defining the tool interface), and
- add new views for more focused queries.

The capability of manipulating tags for information, as well as the chunks that are tagged, distinguishes the tuner from tools provided by typical data management systems.

- Information extraction tools, which are built on top of the information base, which is organized according to methods. These tools essentially exploit the data abstraction consisting of project information organized using the concept tags. These tags are used for the querying facilities.
- Existing tools can be incorporated under TRIAD using a view extractor (to extract the view of the information base for the tool) and an output distributor (to deposit the output of the tool back into the information base). For example, to integrate an interface consistency checker tool, we could extract the appropriate interfaces descriptions from the database and change it to the format expected by the interface consistency checker, have the tool do its processing of the data, and take the output data from the tool and place it in the appropriate places in the data base.

For a typical environment to be practical for all phases of a project life cycle, that environment must support the three types of activities suggested by Li in [64]. These activities are closely related to the software engineering tasks discussed in the previous chapter and include:

BOOK-FORM-2	Chapter	Form-use-#[2]
Title: Introduction		
Introduction: History of the use of methods and their application to a software environment system.		
{3} Section [more?]: Current Problems in Design Methods		Form-use-#[11]
{3} Section [more?]: General Requirements for Improvement		Form-use-#[12]
{3} Section [more?]: Contribution		Form-use-#[13]

BOOK-FORM-3	Section	Form-use-#[11]
Introduction: VLSI design domain have many methods. They fall into two basic classes: flat and type-based.		
Main Point [more?]: New method are based on problems with old methods.		
Main Point [more?]: Some of the management methods have nothing to do with VLSI design.		
Main Point [more?]: New methods are hard to integrate into an existing design system.		
Conclusion [more?]: Present system do not allow for easy integration of experience learned from the use of a method.		
Conclusion [more?]: If a system had a knowledge base of method, easy integration of experience learned from the use of that method would be possible.		

Figure 7: Instantiation of the Chapter and Section Forms

and 7. These instantiations of forms are based on the structure of this dissertation.

Each of the instantiated filled forms has a form-use-# in the upper right corner of the form. This number is used to link the form with an entry in its parent form. A form with the form-use-# of 1 is the root form of the form tree (see figure 6). Certain entries of a form also have form-use-#. This number is the number of the filled form that refines or expands the concept presented by this entry. For example, the third entry in the Book Form (see figure 6) has a form-use-# of 2 which links it to the Chapter Form in figure 7. Linking the forms like this allows the user to view a single instantiation of a concept, all instantiations of a concept, or just a subset of all the instantiations of a concept.

2.1.3. Architecture of the TRIAD Meta System

A good software environment needs three fundamental components - a global monitor, a collection of tool fragments, and an information base [91, 54]. These components take on added functionality in TRIAD's method-based approach. In addition, there is a new unique, component that is called the knowledge base of methods (see figure 2). The four TRIAD components encompass the following collection of tool fragments.

- A tuner tool which allows the users to interactively create the internal representation of the methods (i.e. attributed concept grammar) which comprise the knowledge base of methods. The tuner can...
 - change tags for the concept grammar (i.e. the meta-data),

three tags in blank form 1 namely: Title, Author, and Chapter. The flag [more?] means that this tag and entry is repeatable. The flag {2} means that this entry has a concept that can be further refined by using blank form number 2.

BOOK-FORM-1	Book	Form-use-# [1]
<hr/>		
Title: A Meta System for Generating Software Engineering Environments		
<hr/>		
Author [more?]: Walter L. McKnight		
<hr/>		
{2}	Chapter [more?]: Introduction	Form-use-# [2]
<hr/>		
{2}	Chapter [more?]: Current Research	Form-use-# [3]
<hr/>		
{2}	Chapter [more?]: Method and Methodology	Form-use-# [4]
<hr/>		
{2}	Chapter [more?]: Attributed Grammar Forms	Form-use-# [5]
<hr/>		
{2}	Chapter [more?]: TRIAD and the Grammar Form Model	Form-use-# [6]
<hr/>		
{2}	Chapter [more?]: Tuner	Form-use-# [7]
<hr/>		
{2}	Chapter [more?]: Using the Tuner for a VLSI Method	Form-use-# [8]
<hr/>		
{2}	Chapter [more?]: Using the Tuner for a Virtual Interface Method	Form-use-# [9]
<hr/>		
{2}	Chapter [more?]: Conclusions and Future Work	Form-use-# [10]
<hr/>		

Figure 6: An Instantiation of the Book Form

After selecting a set of project oriented blank forms the user then edits instances of the blank forms to create filled forms which creates a hierarchically organized document called the form tree. A sample of filled forms using our example concept grammar is shown in figures 6

BOOK-FORM-1	Book	Form-use-# <input type="checkbox"/>
<hr/>		
Title:		
<hr/>		
Author [more?]:		
<hr/>		
{2} Chapter [more?]:	Form-use-# <input type="checkbox"/>	
<hr/>		
BOOK-FORM-2	Chapter	Form-use-# <input type="checkbox"/>
<hr/>		
Title:		
<hr/>		
Introduction:		
<hr/>		
{3} Section [more?]:	Form-use-# <input type="checkbox"/>	
<hr/>		
BOOK-FORM-3	Section	Form-use-# <input type="checkbox"/>
<hr/>		
Introduction:		
<hr/>		
Main Point [more?]:		
<hr/>		
Conclusion [more?]:		
<hr/>		

Figure 5: Blank forms for the Book Method

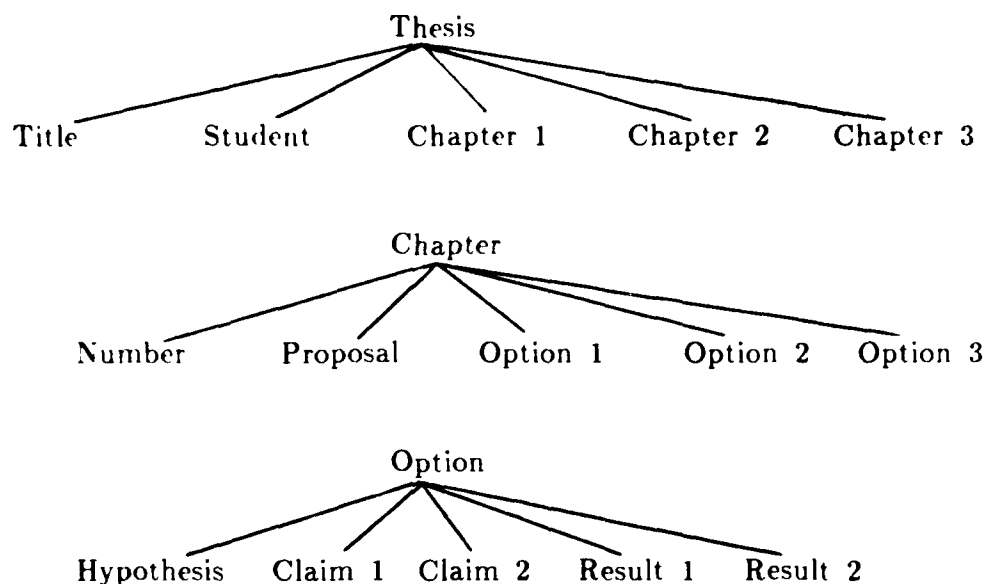


Figure 4: Interpretation Tree for a Book Grammar

A set of blank forms are designed (perhaps by the project manager) to guide the user into following certain methods. Each blank form represents related concepts (productions) of the underlying concept grammar form. A form contains tags which generally describe what is to be in the blanks. The annotations underneath some blanks (called help information) provides information on how to fill in the blank. The text used to fill in a blank is called an entry. The entries in a form must be filled in using some notation, be it English language text, programming language source code, compiled code, or graphical symbols. The form based approach is independent of any specific language. The blank forms which correspond to the concept grammar of our previous example are shown in Figure 5.

Notice that each form has a number in the upper left corner of the form. This number is used for selecting a form. For example, the form number for the Book blank form is number 1 (see figure 5). There are

2. $\langle \text{chapter} \rangle ::= \langle \text{title} \rangle \langle \text{introduction} \rangle \langle \text{section} \rangle^*$

3. $\langle \text{section} \rangle ::= \langle \text{introduction} \rangle \langle \text{main-points} \rangle^* \langle \text{conclusions} \rangle^*$

The concept tree would be as shown in figure 3. One interpretation of the concept tree could be the interpretation tree shown in figure 4.

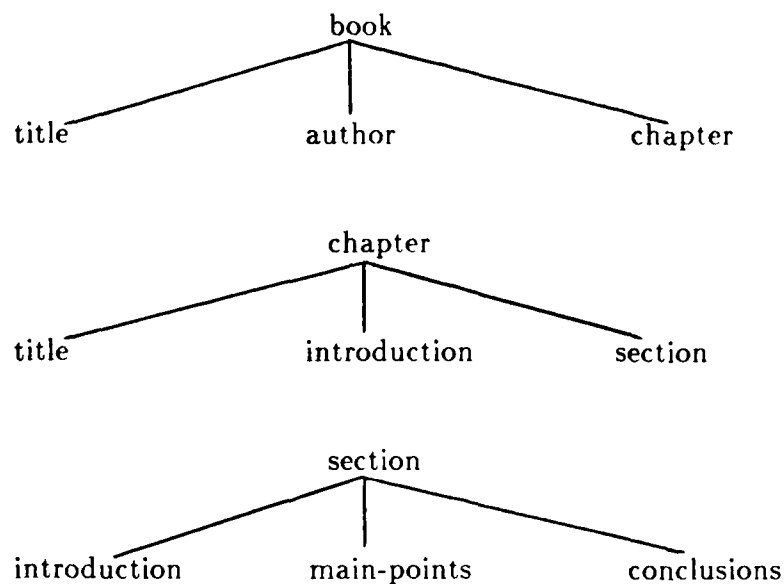


Figure 3: Concept Tree for a Book Grammar

2.1.2. User's View of TRIAD

The user's primary interest is in developing, viewing, and manipulating integrated project information. As far as the user is concerned, the project information base contains hierarchies of forms - blank forms and filled forms [60]. The model underlying the hierarchically related forms is an attributed grammar form, though this is not apparent to the user. The user fills in the blank forms (which become filled forms). Tools can be invoked which will manipulate project information contained in the filled forms. The user can also browse through the filled forms that make up the project information tree.

usually share a common intermediate representation of the program and present a consistent user interface. The goal of integrating a programming environment is to build a tool that does not require the programmer to perform mental context switches, for example, between modifying a program and debugging it. MAGPIE has used as its primary goal, this idea of an integrated programming environment.

MAGPIE addresses only the coding and testing phases of the project life cycle. The coding and testing phases are so much interwound that users are not aware that they are even in the testing phase. In MAGPIE, the user does not need to control, or even initiate the compilation process. Static errors are detected and highlighted in the source code as the program is edited; code generation occurs in the background and is initiated automatically after a portion of the code is modified. MAGPIE supports the Pascal programming language. It uses multiple windows to give different views of the project, but not in the same sense that PECAN does. There is no generator for MAGPIE because the programming environment is for only one language. Because the user is not supposed to make context switches, new tools are rare for MAGPIE and would have to be integrated into the code of MAGPIE directly.

ALOE editors are subsystems of the GANDALF system. ALOE editors are built using a structured editor called ALOEGEN. ALOE is a template based editor in that it provides templates for all productions in the abstract syntax tree and does not automatically allow parsing. CPS provides templates down to the expression level and parses anything lower. It is possible, however, for users to write their own parsers and to use it for limited text editing. Since ALOE is a subsystem, we can

discuss what phases it handles in the project life cycle because other subsystems take care of the phases not covered by ALOE. Of most concern to us is the ALOEGEN system, as our tuner can be viewed as an ALOEGEN system.

ALOEGEN takes a language description for actual language which some ALOE will use to structurally manipulate an abstract syntax tree. Each description contains:

- the description of the abstract syntax of the language,
- the display interface called the unparsing schemes,
- the database interface, and
- the interface to action routines.

Originally, their action routines were very general in nature which caused consistency problems with the database. One action routine could undo what another action routine had done and there was no control on the order in which these action routines were called. In their latest version [2], they have gone to more restricted action routines which are more like our procedural components.

As with the generator for CPS, our tuner has taken some ideas from ALOEGEN. However, there are major differences which include:

- In ALOEGEN, the attributes can only be referenced through the node extension. In the tuner, attributes are referenced one of three ways with each way given a priority. The highest priority is given through the current node; the next is

given through direct links with the attribute; and the lowest is through the root node of the tree. All global attributes are attached to the root node.

- In ALOEGEN, the action routines must take care of the firing condition. In the tuner, the firing condition is attached to the action name. All actions under a certain firing condition must be included in the same procedural component.
- In ALOEGEN, the action routines are compiled with the language description. In the tuner, the procedural components are not interpreted until they are needed so that binding to the attributes can be delayed to the latest possible time.

GANDALF is a project development system that addresses some of the issues of programming-in-the-large. Some of the support that it provides is:

- System Version Control support which helps describe the interfaces and composition of subsystems in order to permit automatic generation of system versions.
- Project Management support which helps control the development process so that programmers can make changes to a project in an orderly fashion.
- Incremental Program Construction support which helps the individual programmer develop a single program in relative isolation from other programmers.

At present, complete support for the project life cycle is not available, as

the requirements and design phases are not supported. However, the GANDALF system has done a better job in the support of the other phases of the project life cycle than any other system. TRIAD gives a more management oriented approach to project management than GANDALF in that GANDALF project management control is only controlled message passing. Each of the support systems of GANDALF is controlled by an ALOE system which has already been addressed.

ARCTURUS is an Ada project development system. Its main emphasis is on the tight coupling of tools, effective command and editing concepts, and the combined use of interpretive and compiled execution. ARCTURUS is not complete however. Practical, mature environments for a language like Ada must also support development, analysis, testing, and debugging of concurrent programs. ARCTURUS is like the SUPPORT system in that it has a design language and uses stepwise refinement of the design to come up with the executable code. ARCTURUS is driven by one method and appears to lack any management method. Because ARCTURUS has only one language, a generator is not in the system. All new features must be compiled and integrated directly into the system. ARCTURUS has the following features:

- templated based with three ways to fill in the holes; another template, known identifiers with completion capability, and normal text,
- interactive techniques such as breakpoints, tracing, interruption and resumption of computations, queries, and pretty-printing,
- tight integration - in the form of sharing a few large, complex

data structures between the tools - reducing the conceptual overhead in tool-switching and execution efficiency,

- catalogue powerful user-interface techniques and move them out of individual tool interfaces into a common interface.

SAGA is a project development system that is addressing the issues associated with managing the project life cycle for small to medium sized software development project involving a team of up to twenty members. Each phase of the project life cycle is described by a machine-recognizable language which does not necessarily have to be executable. These languages are generated by a language generator that is similar to a parser-generator. Any parser generating system may be used if the resulting parser and its tables can support the functions required by the interface to the editors. Input to the parser generator consists of the formal description of the grammar of the language, the formal description of the lexical representations of the tokens, and the semantic evaluation information in the form of executable code fragments. The parser generator produces parse tables and associated information which is combined with the parser generator dependent library routines and the common editor object code to produce an editor for a particular language. Unlike the tuner, which uses one parser generator, SAGA could use a different parser generator for each phase of the project life cycle.

Also included in the SAGA project is a source code control subsystem that records the modifications made to parse trees and the associated information maintained by the language-oriented tools. It also includes the Notesfile system which is used to document the system as well as technical discussions, product reviews, hardware and software bugs and

fixes, agendas and minutes, design and specification documents, appointments, news, and mail.

Another project development system that is similar to TRIAD is the ISDE system. The principle distinguishing features of their approach are:

- the derivation of an environment for a specific language as an instantiation of a language-independent meta-environment,
- the definition and implementation of a unique internal representation of programs to allow a complete integration among the tools, and
- a high degree of interaction to allow the user to incrementally define and analyze programs.

Their approach is similar to TRIAD in the sense that each environment is built from a generic environment. However, they have gone with tight tool integration where TRIAD has basically gone with loose tool integration. They do address issues of programming-in-the-large and most phases of the project life cycle. They have a generator for their system but no information was available to compare it with the tuner.

A summary comparison of these systems is shown in figure 8. Not included in this comparison are some unique systems which have taken different approaches to programming environments. The OMEGA system [79, 65, 66] implements a programming environment on a relational database. The IDEOSY system [38] is a graph-oriented language programming environment.

Chapter 3

Methods and Methodology

Examination of a wide variety of industrial case studies has shown that the evolution of numerous software engineering methods is based on deep differences among various project types in various industrial settings. Fundamentally, a software engineering method organizes and manipulates software-related information (requirements, specifications, management plans, design, code, etc.). The automation and enforcement of a software engineering method could be handled by a software engineering environment. The role of a software engineering environment is to provide support to view the software-related information, manipulate it, and apply tools to it [33].

Methods have evolved for producing software within specific domains or project types. These methods have been based on experience gained from working with projects in those domains. Experience often suggests changes for the method. This has motivated research in meta software environments which provide support for an environment that is not hard wired to the details of a specific, pre-determined method. Rather, the meta environment gets customized to any method from a collection of methods prescribed by the project manager and provides more focused support for the needs of a particular user community.

The unique power of the TRIAD meta environment is attributable to a separate knowledge base of methods. Each method is represented using

an extended attributed grammar form. Anyone of these representations can drive the meta environment. This provides the following functionality

- The ability to select a method most appropriate for the project and to use the method description to customize the meta environment and, thus, the support it provides for the project.
- The ability to add new, judiciously designed methods to the knowledge base which provide integrated support for a wide spectrum of software engineering tasks.
- The ability to tune methods to adapt to changing user needs. A method can be modified to reflect a user group's experience with a method in a project.

In this chapter we will look at methods and how they have evolved. In section 3.1 we will define a method. We will also see how some of the current methods support various phases of the project life cycle. Not all methods need to support the project life cycle. In section 3.2 we will discuss the evolution of a method. We will also discuss in that section how tools to automate parts of a method can evolve.

3.1. What is a Method?

Method consists entirely in properly ordering
and arranging the things to which we
should pay attention. **Descartes**

According to the dictionary, a method is "an orderly procedure or way of accomplishing a task". We define a component method to be a

procedural way to implement some technology, to enforce some design practice, or to enforce some management constraint. Then our definition of a method consists of component methods, techniques for managing the application of the component methods, and tools for automating the support of these component methods. A component method is what others have referred to as a methodology. Technically, the dictionary meaning of methodology would suggest the study of methods and how they evolve. For example, Jackson methodology [49] would be called the Jackson method by our definition.

The Jackson method suggests, among other things, that the first task is identifying the input and output structures of the files, keeping in mind the steps required to solve the problem. It also describes the next task, which is to develop the program body itself. Whereas the Jackson method is oriented towards the product (the program), other methods like the Mead Conway method [73] are oriented towards the process of developing the product. Typically, a phased process like the software engineering process, with tasks and schedules for each phase, is used with appropriate checkpoints, walk-throughs and reviews. Note process tasks (development-in-the-large tasks and project management tasks) dominate in larger projects. Currently used methods are oriented towards a limited subset of software engineering tasks. They do not provide uniform, integrated support for developing the product using a systematic process to the degree that is possible.

A method should not be a collection of instructions in which the user's success depends on his choosing the right set of instructions. Instead, a method should provide the user with precise guidelines as to when and how to apply each instruction. This will allow the user to follow a

systematic approach to solving the problem. Most of the software engineering methods are imprecisely described and can not be used to enforce systematic problem solving.

Many methods have been developed. Some of the methods associated with various phases of the project life cycle are shown in figure 9. Other methods have been developed for designing VLSI chips. These methods all provide good design techniques based on some principles or experience. They are all described using natural languages, but not necessarily procedural languages. It is important that a method be precisely described so that different users apply the method in a standard way.

- * Planning Phase
 - Determine Scope of Project
PERT and CMP - [108]
 - Requirements Definition and Analysis
SADT - [87, 88]
PSL/PSA - [101]
SREM - [18]
- * Development Phase
 - Preliminary Design
Jackson - [49]
Structured design - [110]
 - Detail Design
HIPO - [99]
 - Coding
Data abstraction - [67]
 - Testing
DAVE - [77]
DISSECT - [47]
- * Maintenance Phase
 - Enhancements
 - Problem Correction

Figure 9: Methods For the Project Life Cycle

Methods are problem domain dependent. For example, the Jackson method works nicely for the database processing domain but is totally impractical in the real-time processing domain. A method is usually designed based on the understanding of a problem domain, the experience accumulated in the past in solving problems in that domain, and the algorithms available. When a problem domain becomes more specific, problems in that domain are better understood and therefore a more detailed and precise method can evolve.

3.2. Steps in the Evolution of Methods and Their Tools

Each problem that I solve becomes a rule which
served afterwards to solve other
problems. **Descartes**

Methods and their related tools are not created overnight; an evolutionary process is entailed. Take for example, the Jackson method. It was developed in the early 1970's for the data processing domain. Logrippo designed an automatic tool for generating skeletal code based on the Jackson method in the early 1980's [68]. This automatic tool used an attributed grammar to capture the structure of the files to generate the control structure of code. In 1984, Shubra designed a more comprehensive system of seventeen templates based on Jackson method [92]. Each template provides more detailed support for a sub-class of data processing problems. Each of the above enhancements was built on previous experience with the Jackson method and with its associated tools.

There are two related ways in which this evolution takes place. One is the degree of automated support for the method and the other is the further refinement of the method itself to reflect domain experience. The

degrees of automated information handling support made possible by enhancing a method are indicated below:

1. A method could provide guidelines and/or notation for organizing information so that appropriate information is extractable and visible using simple data base techniques.
2. In addition to the above, a method could provide guidelines for analyzing and correlating information in a more global way.
3. Finally, in addition to 1 and 2 above, a method could also simplify problem solving by regulating and monitoring tasks.

Degree 1 is accomplished by chunking project information [107]. Often, chunking takes on the form of a hierarchical organization of information to reflect the refinement process. Continuing with the Jackson method, the viewing of this information could be accomplished by providing a form interface to a data base. Forms could display the meta-data; the record description (keys, format, etc.), and serve as an interface to formulate queries. For example, a key called (input file(s)) could tag the chunk of information containing all the input files. A query could then be designed to extract information tagged by (input file(s)). Information could be extracted by doing some sort of incremental search through the data base described by the meta-data.

Degree 2 is accomplished when tags are used more effectively. The semantic content of tags and the relationships between the tags could convey certain concepts underlying a method for organizing information. This layer built on top of a data base allows the user to formulate more

effective queries. With the Jackson method, in an example suggested by Soni [96], we could define a concept (based on tags and the relations between them) for $\langle \text{input} \rangle$. This concept could include a structure as shown in figure 10. This concept is...

$$\begin{aligned} \langle \text{file} \rangle &::= \langle \text{structure} \rangle^* \\ \langle \text{structure} \rangle &::= \langle \text{structure} \rangle^* \\ &\quad | \langle \text{null} \rangle \end{aligned}$$

An interpretation of this concept (also shown in figure 10) would produce the following grammar:

$$\begin{aligned} \langle \text{transaction file} \rangle &::= \langle \text{customer} \rangle^* \\ \langle \text{customer} \rangle &::= \langle \text{matched customer} \rangle^0 \langle \text{unmatched customer} \rangle^0 \\ \langle \text{matched customer} \rangle &::= \langle \text{valid transaction} \rangle^* \\ \langle \text{unmatched customer} \rangle &::= \langle \text{valid transactions} \rangle^* \\ \langle \text{valid transactions} \rangle &::= \langle \text{transaction record} \rangle \end{aligned}$$

This concept could provide the capability of accessing any view by very simple queries like:

- Display $\langle \text{input} \rangle$
- Display $\langle \text{file structure} \rangle$ of $\langle \text{input} \rangle$

To do the same without the concept based use of tags, we would have to have the following queries:

- Display (then list the fields of $\langle \text{input} \rangle$)
- Display (then list the fields of the $\langle \text{file structure} \rangle$ of $\langle \text{input} \rangle$)

Even for more complex examples, concept based queries remain fairly simple, as compared with the queries on a data base (such as a relational one) which become very tedious.

Degree 3 is accomplished when views are extracted under control logic external to the concept based information base. Such logic, called global controller logic, can extract views for tools supporting the method, can take the output of tools and place it back in the concept information base, can invoke different tools and execute different queries to automate tasks suggested by the method. Furthermore, procedures and attributes can be associated with the concept tags. These procedural components can also be fired by the global control logic in a systematic way. For example, using the Jackson method again, if we incorporate the ideas presented by Kiper [57], we can integrate a Cobol compiler that is fed a view of the concept information base consisting of the Cobol program string. The global controller has knowledge of the method and can systematically fire procedures, attached to tags, to extract the code from the chunk of text associated with the tag. Furthermore, a global controller can also ensure (using attributes) that the body of a program does not get developed until the input and output structures are defined according to the Jackson method.

As experience with a method builds up, the method support can be progressively automated from 1 to 3 above. It is possible to modify the method in systematic ways as it is being applied.

Let us suppose that we have a context-free grammar G . A context-free grammar is ambiguous if there is some string $\omega \in L(G)$ (where $L(G)$ is the language generated by G) that has at least two distinct derivation trees. Ambiguity is a property of natural interest in the study of context-free grammars, since if a string can be derived in two different ways the intended meaning of the string may be in doubt. An example of an ambiguous context-free grammar is the following:

$G = (v, \theta, P, \sigma)$ where

$$v = \{\text{expression, term, } x, y, +, \times\}$$

$$\theta = \{x, y, +, \times\}$$

$$(v - \theta) = \{\text{expression, term}\}$$

$$P = \{P_1 : \langle \text{expression} \rangle ::= \langle \text{term} \rangle$$

$$P_2 : \langle \text{expression} \rangle ::= \langle \text{expression} \rangle \langle + \rangle \langle \text{expression} \rangle$$

$$P_3 : \langle \text{expression} \rangle ::= \langle \text{expression} \rangle \langle \times \rangle \langle \text{expression} \rangle$$

$$P_4 : \langle \text{term} \rangle ::= \langle x \rangle$$

$$P_5 : \langle \text{term} \rangle ::= \langle y \rangle \}$$

$$\sigma = \langle \text{expression} \rangle$$

For example, $x \times y + x$ has the two distinct derivation trees shown in Figure 12. The reason this grammar would be impractical (as a way of describing even a subset of the arithmetic expressions of a real programming language) is that it gives no clue as to the intended way of computing an expression such as $x \times y + x$: should we multiple $x \times y$ first or add $y + x$ first?

Unfortunately, as desirable as the lack of ambiguity might seem to be,

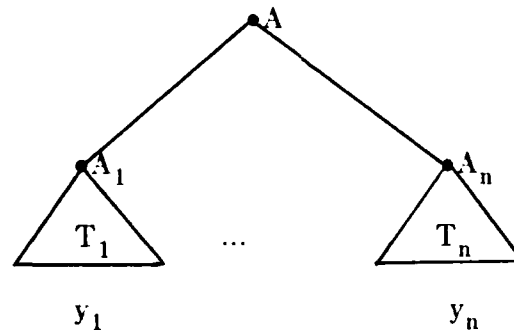


is a derivation tree; its root is the node labeled A , its only leaf is the node labeled λ , and its yield is λ .

3. If



are derivation trees ($n \geq 1$) with roots labeled A_1, \dots, A_n , respectively, and with yields y_1, \dots, y_n , and $P_i: A ::= A_1 \dots A_n$ is a production in P , then



is a derivation tree. Its root is the new node labeled A , its leaves are the leaves of T_1, \dots, T_n , and its yield is y_1, \dots, y_n .

4. Nothing else is a derivation tree.

Now that we have defined what a derivation and a derivation tree are we can now look at some properties of context-free grammars. The first property we want to look at is ambiguity.

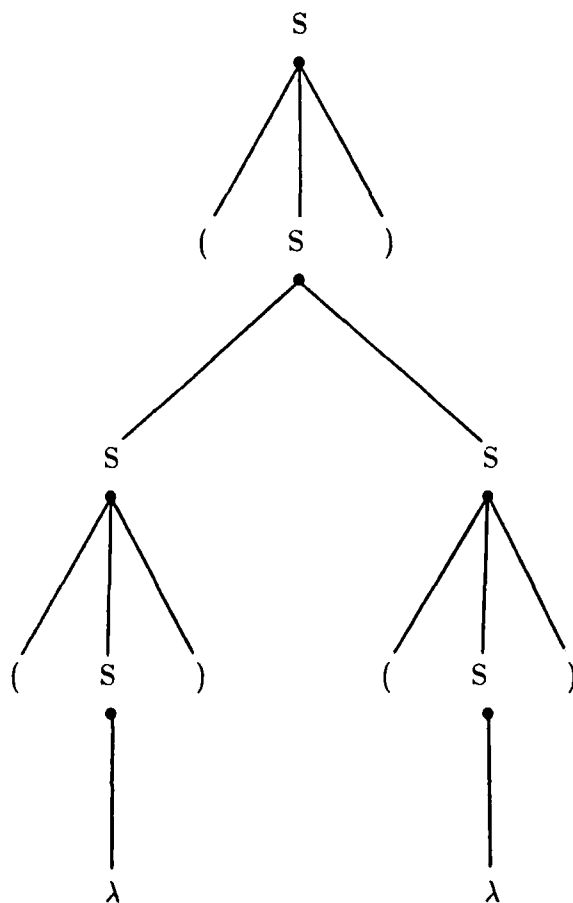


Figure 11: Derivation of $((\lambda))$

More generally, for any context-free grammar $G = (v, \theta, P, \sigma)$, we define its derivation trees and their roots, leaves, and yield, as follows.

1. $\bullet A$

This is a derivation tree for each $A \in v$. The single node of this parse tree is the root and a leaf. The yield of this derivation tree is A .

2. If $P_i: A ::= \lambda$ is a production in P , then

For an example of a derivation let us use the example of the balanced parenthesis and see what the derivation of $((()))$ would be:

$G = (v, \theta, P, \sigma)$ where

$$v = \{S, (,)\}$$

$$\theta = \{(,)\}$$

$$(v - \theta) = \{S\}$$

$$P = \{P_1 : \langle S \rangle ::= \langle \lambda \rangle$$

$$P_2 : \langle S \rangle ::= \langle S \rangle \langle S \rangle$$

$$P_3 : \langle S \rangle ::= \langle (\rangle \langle S \rangle \langle) \rangle \}$$

$$\sigma = \langle S \rangle$$

A derivation of $((()))$ is

$$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (()S) \Rightarrow (()S) \Rightarrow (()S) \Rightarrow (()())$$

Notice that the derivation of $((()))$ could also have been,

$$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow (S(S)) \Rightarrow (S()) \Rightarrow ((S)()) \Rightarrow (()())$$

In fact there are a total of six possible ways of deriving $((()))$. However, these six derivations are, in a sense, the same. The rules used are the same, and they are applied at the same places in the intermediate string. The only difference is in the order in which the occurrences of the non-terminal S are replaced. All the derivations can be pictured as in Figure 11.

We call such a picture a derivation tree. The points are called nodes; the topmost node is called the root, and the nodes along the bottom are called leaves. By concatenating the labels of the leaves from left to right, we obtain the derived string, which is called the yield of the derivation tree.

$$v = \{\text{sentence}, (,)\}$$

$$\theta = \{(,)\}$$

$$(v - \theta) = \{\text{sentence}\}$$

$$P = \{P_1 : \langle \text{sentence} \rangle ::= \langle \lambda \rangle$$

$$P_2 : \langle \text{sentence} \rangle ::= \langle \text{sentence} \rangle \langle \text{sentence} \rangle$$

$$P_3 : \langle \text{sentence} \rangle ::= \langle (\rangle \langle \text{sentence} \rangle \langle) \rangle \}$$

$$\sigma = \langle \text{sentence} \rangle$$

4.3. Generative Grammars

Now that we have defined context-free grammars, we want to investigate some of their properties. In order to do so we need to first define a derivation. Then we will look at a representation of a derivation called a derivation tree. Next we will look at two properties of grammars namely, ambiguity and non-determinism. Finally, we will look at three ways that context-free grammars are used and what problems are associated with each of these uses.

Let G be a formal grammar. A string of symbols in $v^* \cup \lambda$ is known as a sentential form. If $\alpha \rightarrow \beta$ is a production of G and $\omega = \delta\alpha\psi$ and $\omega' = \delta\beta\psi$ are sentential forms, we say that ω' is immediately derived from ω in G , and we indicate this relation by writing $\omega \Rightarrow \omega'$. If $\omega_1, \omega_2, \dots, \omega_n$ is a sequence of sentential forms such that $\omega_1 \Rightarrow \omega_2 \Rightarrow \dots \Rightarrow \omega_n$, we say that ω_n is derivable from ω_1 and indicate this relation by writing $\omega_1 \stackrel{*}{\Rightarrow} \omega_n$. The sequence $\omega_1, \omega_2, \dots, \omega_n$ is called a derivation of ω_n from ω_1 according to G .

Right hand expressions for context-free grammars are called phrases. Like with regular grammars the symbol A in the left hand expressions can only be a generating symbol. Both δ and ψ must be empty strings. This class of grammars gets its name because replacing the non-terminal A in the left hand expression is independent of any adjacent symbols; that is, it is independent of the context.

Formally, a context-free grammar G is a quadruple (v, θ, P, σ) where

- $v \subset V$ is a vocabulary.
- $\theta \subset v$ is the terminal vocabulary (where $\theta \subseteq \Theta$).
- $(v - \theta)$ is the non-terminal vocabulary (where $(v - \theta) \subset (V - \Theta)$).
- $P \subset (v - \theta) \times v^*$ is a finite set of production rules such that for each $P_i \in P$, P_i has the form:

$$P_i : A ::= B_1 B_2 \cdots B_n$$

with finite $n > 0$, where $A \in (v - \theta)$ and each $B_j \in v$. P_i is the production number, A is the left-hand side of the production, and each B_j is a member of the right-hand side of the production.

- σ is a distinguished member of $(v - \theta)$ called the start symbol.

For an example of a context-free grammar, let's take the grammar that determines if the parenthesis in a sentence are balanced. The grammar for this problem is:

$$G = (v, \theta, P, \sigma) \text{ where}$$

1. concatenation: $\alpha_1 \alpha_2 \cdots \alpha_n$
2. alternative: $\alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$
3. repetition of an expression: α^*

An example of a regular grammar is the following grammar which verifies that a string is a real number:

$$\Delta = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, ., \text{digit}, \text{decimal point}, \text{sign}, \text{number}\}$$

$$\theta = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .\}$$

$$(\Delta - \theta) = \{\text{digit}, \text{decimal point}, \text{sign}, \text{number}\}$$

$$P = \{\langle \text{number} \rangle ::= \langle \text{sign} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle^* \langle \text{decimal point} \rangle \langle \text{digit} \rangle^*$$

$$\mid \langle \text{sign} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle^*$$

$$\mid \langle \text{digit} \rangle \langle \text{digit} \rangle^* \langle \text{decimal point} \rangle \langle \text{digit} \rangle^*$$

$$\mid \langle \text{digit} \rangle \langle \text{digit} \rangle^*$$

$$\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$\langle \text{sign} \rangle ::= + \mid -$$

$$\langle \text{decimal point} \rangle ::= . \}$$

$$\sigma = \langle \text{number} \rangle$$

A less restrictive class of grammars is the context-free grammar class. In this grammar the general format of productions are:

$$\begin{aligned} A &\Rightarrow \omega \text{ where} \\ A &\in (V - \Theta) \cup \{\sigma\} \\ \omega &\in V^* - \{\lambda\} \end{aligned}$$

- Right hand expression. The right hand expression is a collection of symbols which will be generated by the grammar when the symbol A of the left hand expression is reduced or generated. Its format is $\delta\gamma\psi$. The collection of symbols γ is called the reduced or generated string depending on the class of the grammar.

By restricting the format that productions of a grammar are permitted to take, we can divide grammars into classes of grammars. The class of grammars with the most restricted format for its productions is called regular grammars. The general form of productions for regular grammars are either:

$$\begin{aligned} A &\Rightarrow \alpha B \\ A &\Rightarrow \alpha \quad \text{where} \\ A &\in (V - \Theta) \cup \{\sigma\} \\ B &\in (V - \Theta) \\ \alpha &\in \Theta \end{aligned}$$

or

$$\begin{aligned} A &\Rightarrow B\alpha \\ A &\Rightarrow \alpha \quad \text{where} \\ A &\in (V - \Theta) \cup \{\sigma\} \\ B &\in (V - \Theta) \\ \alpha &\in \Theta \end{aligned}$$

Right hand expressions for regular grammars are called regular expressions. The symbol A in the left hand expression for regular grammars can only be a generating symbol. Both δ and ψ must be empty strings. Regular expressions are formed from the non-terminal and terminal symbols of the grammar, as well as the empty string λ , by applying the following formation rules recursively any number of times:

$$\langle \text{object} \rangle ::= \langle \text{man} \rangle | \langle \text{girl} \rangle | \langle \text{ball} \rangle | \langle \text{hat} \rangle$$

and that our start symbol is the non-terminal symbol $\{\text{sentence}\}$. A few of the syntactically correct sentences that we could generate are:

man hit ball

girl throw hat

hat throw man

ball hit girl

ball throw man

As one can readily see, what might be syntactically correct might not be semantically correct.

Formally, a grammar is a quadruple $(\Delta, \theta, P, \sigma)$ where

- $\Delta \subset V$ is a vocabulary.
- $\theta \subset \Delta$ ($\theta \subseteq \Theta$) is the terminal vocabulary.
- $(\Delta - \theta) \subset (V - \Theta)$ is the non-terminal vocabulary.
- P is a finite set of production rules.
- σ is a distinguished member of $(\Delta - \theta)$ called the starting symbol.

Production rules have two basic parts, namely:

- Left hand expression. The left hand expression is a collection of symbols already generated by the grammar. Its format is $\delta A \psi$. The symbol A is called the reducing or generating symbol depending on the class of the grammar.

vocabulary Θ . Note that Θ can only be a subset of V since Θ is finite and V is infinite.

2. Non-terminal symbols. Non-terminal symbols are a collection of phrase denoting symbols of the grammar. These symbols never appear in the generated sentences but are used as intermediate phrases for the final generation of the sentences of the language. We will call this vocabulary Φ . Note again that Θ can only be a subset of V and that Θ and Φ have to be disjoint.
3. Production rules. Productions rules are a collection of grammatical rules which tell how to go from one intermediate phrase to a collection of phrases and/or a collection of terminal symbols. We will call this collection of production rules Ω .
4. Starting symbol. The starting symbol is the starting point for applying the productions. We will call this symbol σ . Note that σ does not necessarily have to be element of Θ , but it could be.

To continue with our example of the English language, let us suppose that our set of terminal symbols is {man, hit, ball, throw, girl, hat}, that our set of non-terminal symbols is {sentence, subject, verb, object}, and that our collection of productions is:

$$\langle \text{sentence} \rangle ::= \langle \text{subject} \rangle \langle \text{verb} \rangle \langle \text{object} \rangle$$

$$\langle \text{subject} \rangle ::= \langle \text{man} \rangle | \langle \text{girl} \rangle | \langle \text{ball} \rangle | \langle \text{hat} \rangle$$

$$\langle \text{verb} \rangle ::= \langle \text{hit} \rangle | \langle \text{throw} \rangle$$

introducing the concept of families of grammars known as grammar forms. In section 4.7 we will add semantic meaning to our grammar forms by introducing attributed grammar forms. For those who have a good foundation in grammar theory, they can skip the material up to section 4.6 without any problem.

4.2. Context Free Grammars

In the last section we informally introduced the concept of a grammar. In this section we will formally define a grammar. We will look at the various parts of a grammar. Finally, we will introduce two classes of grammars; regular grammars and context-free grammars.

An alphabet Σ is a finite non-empty set of symbols or letters. A string x over Σ is a finite, possibly empty, sequence of symbols from Σ . The empty string is denoted by λ . A vocabulary V is an infinite set of symbols or letters. For example, in the English language the alphabet consist of 71 symbols (26 small letters, 26 capital letters, 10 numeral symbols, and 9 punctuation symbols). A subset of the vocabulary for the English language is what is typically found in an unabridged English dictionary. Throughout this chapter we will use the notion of vocabulary instead of what is typically used, the alphabet.

A grammar has four major components:

1. Terminal symbols. Terminal symbols constitute the vocabulary from which sentences of the described language are constructed. The generation of a sentence through the application of grammatical rules must terminate in a string containing only these terminal symbols. We will call this

programming language is usually considered to be any string of these symbols that represent a complete program. A satisfactory grammar for a programming language should permit one to determine by a mechanical procedure whether an arbitrary sequence of symbols is a "well-formed" program.

Again, according to the dictionary, a grammar is "the system of rules implicit in a language viewed as a mechanism for generating all sentences possible in that language." From a computer science point of view, we would call this definition of grammar "the syntax of a language." A considerable amount of research has been conducted regarding the structure of programming languages. Several mathematical models have been defined and many programming languages have been built on top of these models. Also inherent in grammars is the notion of the meaning given to a syntactically correct sentence of the language. Only recently has there been any concentrated effort on studying programming language semantics.

In this chapter we will look at a class of languages called context-free languages. In section 4.2 we will formally define what a grammar is and then discuss a class of grammars called context-free grammars. In section 4.3 we will look at the problems associated with a subclass of the context-free grammar called context-free generative grammars. In section 4.4 we will extend our notion of grammar by including semantics. Semantics will be introduced by allowing attributes to be defined with the syntax of the grammar. In this section we will define what an attributed grammar is. In section 4.5 we will discuss the simplicity gained with the introduction of attributed grammars with right regular parts. In section 4.6 we will extend our notion of grammars by

Chapter 4

Attributed Grammar Forms

4.1. Introduction

According to the dictionary, a language is "a historically established pattern of words and symbols that offers substantial communication only to the individuals within the culture it was defined." This statement is hardly specific enough to serve as an adequate mathematical description of the term language. Language experts agree that any adequate formalism for natural language must cope with both the structure of the language, known as syntax, and the meaning given to sentences in the language, known as semantics. One can not hope to exhaustively list all possible sentences in a language, for each user of a language is able to speak sentences that have never previously been spoken. Likewise, one cannot hope to understand every sentence uttered or written. Similarly, one cannot hope to write down all possible Pascal or Cobol computer programs, for any user is able to write programs never before written that will run perfectly well on a computer. Thus, the central problem of describing a language is to provide a finite specification for an essentially infinite class of objects.

Given an alphabet V , the set V^* is the totality of all possible strings on V , and a language L on V is an arbitrary subset of V^* . For a computer programming language, the alphabet is the collection of all nondivisible symbols that may appear in a program. A sentence in a

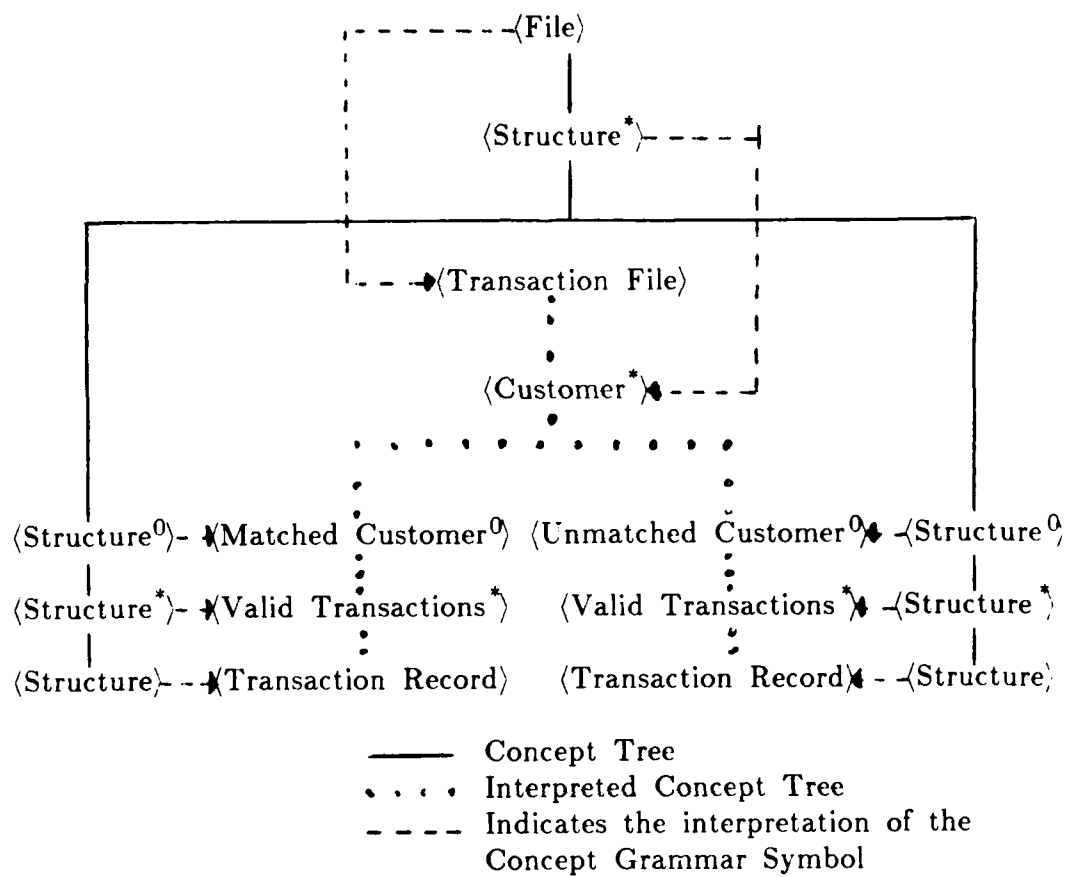
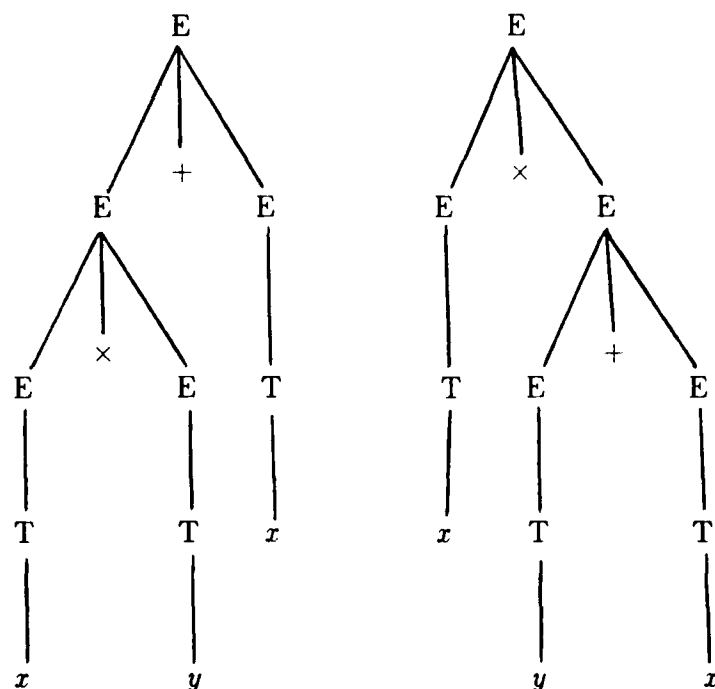


Figure 10: Relationship Between a concept tree and its interpreted concept tree



where $E = \langle \text{expression} \rangle$ and $T = \langle \text{term} \rangle$

Figure 12: Derivation trees of $x \times y + x$

there is no general procedure for telling whether a grammar enjoys this property.

Theorem 1: There is no effective procedure for deciding, given any context-free grammar G , whether G is ambiguous or not.

See [21] for proof of this theorem.

Now that we have seen the problem with ambiguity, let us now turn our attention to our second property: non-determinism. In order to properly define non-determinism, we need to first correlate our ideas of a grammar with that of a computing machine. A computing machine is made up of three parts, namely:

1. finite controller which is often called the central processing unit,
2. input/output device, and
3. memory.

The machine picks up one symbol from the input device and, based on the current state of the finite controller and what is contained in memory, determines what the next state of the finite controller is going to be and what symbol is going to be output on the output device (the output may be nothing at all). The production rules of a grammar tell us how to go from one state of the finite controller to the next. The next input symbol corresponds to the next symbol in the current production being processed. With this symbol and the current production, we are able to determine which production is the next one to be processed.

Non-determinism is essentially the ability to change states in the machine in a way that is only partially determined by the current state and the input symbol. Several possible next states, for a given combination of current state and input symbol, are permitted. In the grammar sense it means we have several choices for the refinement of a symbol. Let us illustrate this with an example common to most programming languages. Let us suppose that our language allows both if-then and if-then-else statements. Then our grammar would include the following production rules:

$$\langle \text{statement} \rangle ::= \langle \text{if-then statement} \rangle$$

$$| \langle \text{if-then-else statement} \rangle$$

$\langle \text{if-then statement} \rangle ::= \langle \text{IF} \rangle \langle \text{boolean exp} \rangle \langle \text{THEN} \rangle \langle \text{statements} \rangle$

$\langle \text{if-then-else statement} \rangle ::= \langle \text{IF} \rangle \langle \text{boolean exp} \rangle \langle \text{THEN} \rangle \langle \text{statements} \rangle$
 $\langle \text{ELSE} \rangle \langle \text{statements} \rangle$

If we are in the state called "statement" and our input symbol is IF, there is no way that we can uniquely determine whether that our next state should be "if-then statement" or "if-then-else statement". We have to make a non-deterministic choice on which state we will go to next.

Unfortunately, though it is desirable to guarantee that a grammar is deterministic, there is no general procedure for telling whether a grammar enjoys this property.

Theorem 2: There is no effective procedure for determining whether a given context-free grammar is deterministic.

See [21] for proof of this theorem.

Now that we have looked at two important properties of context-free grammars and found out what problems they create, let us now turn our attention to three uses of context-free grammars.

These three uses are as acceptors, as transducers, and as generators. In order to discuss these uses we first need to define what a push-down automata is and show its relation to context-free grammars.

Informally, push-down automaton is one of our computing machines we mentioned earlier. This computing machine has unbounded memory in the form of a stack in which we are only allowed to use the top-most element of the stack and the current input symbol in our decision on which state we can next enter. The input device is generally considered to be an input tape where each symbol read takes one place and after

its read, it can not be read again (i.e. the tape only moves in one direction). The output device also is a tape which can hold one symbol in each place on the tape and the tape can only move in one direction.

Formally, a push-down automaton is a sextuple $M = (K, E, \Gamma, \Delta, s, F)$ where

- K is a finite set of states,
- E is an alphabet (the input symbols),
- Γ is an alphabet (the stack symbols),
- $s \in K$ is the initial state,
- $F \subseteq K$ is the set of final states, and
- Δ , the transition function, is a finite subset of $(K \times E^* \times \Gamma^*) \times (K \times \Gamma^*)$

Intuitively, if $((p, u, \beta), (q, \gamma)) \in \Delta$, then M , whenever it is in state p with β at the top of the stack, may read u for the input tape, replacing β by γ on the top of the stack, and enter state q . Such a pair $((p, u, \beta), (q, \gamma))$ is called a transition of M .

A well known theorem of push-down automata theory is:

Theorem 3: The class of languages accepted by push-down automata is exactly the class of context-free languages.

The proof of this theorem can be found in [63]. This means we can think of push-down automata as the physical machine defined by the context-free grammar. Push-down automata and context-free grammars can be used synonymously.

The first use of push-down automata(context-free grammar), is as an acceptor. Basically, a push-down acceptor determines whether an input string to the acceptor is a string in the language described by the push-down acceptor. The push-down acceptor works just like the push-down automaton we described except that the only output is y if the string is accepted and n if the string is not accepted. The properties of ambiguity and non-determinism create problems for acceptors. However, attempts have been made to try to overcome these problems.

When used in a compiler to perform syntax analysis, acceptors are generally called parsers. The problem of ambiguity for arithmetic expressions is partially overcome with precedence rules, but precedence rules are not really a part of the grammar (from a context-free grammar point of view). They are more of a semantic issue and should not be a part of the parser.

One possible way to partially overcome the problem of non-determinism is with a look-ahead parser. In a look-ahead parser, when a symbol is read from the input tape and the parser has several options of which productions to use to refine this symbol, the parser will look ahead up to a certain number of symbols to resolve which production to use. The other possibility is with a back-tracking parser. In this case, the parser arbitrarily picks one production to refine the symbol read in from the input tape and continues execution until either the string is accepted or rejected. If the string is rejected, the parser backs up to the point where it had to make the arbitrary choice and makes another choice. Only after all the choices have been exhausted will the parser indicate that the string is not accepted.

The second use of push-down automata is as a transducer. A push-

down transducer will take the input string and translate it into another string that could be accepted by another push-down automata. A good example of the use of a transducer is the code generator of a compiler. Again the properties of ambiguity and non-determinism create problems for the transducer, even though we do not see these issues addressed with a code generator. The reason these issues are not addressed with code generator is that the string input to the code generator has already been accepted by a parser before it was sent to the code generator.

Another example of a transducer is a parser that outputs a history of what it has done, or one that outputs an intermediate representation of the input string. In either case the same solutions to the problems of ambiguity and non-determinism that were inherent in acceptors are also inherent here. The same approach to overcome these problems are also used.

The third use of push-down automata is as a generator. A push-down generator, basically, generates strings for a context-free language that will always be accepted by a push-down acceptor for that same language. The generator begins in the initial state of the finite controller and keeps generating symbols of the string until it comes to a point in the generation where a choice has to be made. At that point it reads a symbol from the input tape which indicates what choice to make. The output tape contains the generated string which would be accepted by a push-down acceptor for that same language. The properties of ambiguity and non-determinism do not create a problem for the generator.

Generators are not used in compilers. Only with the recent development of programming environments has there been a need or a use for a generator. Generators have been used in some of the

programming environments like ALOE [74], CPS [84, 85], PECAN [82, 83], and Arcturus [98]. All of these systems are template based. When a choice needs to be made to continue the generator, the user is presented with the possibilities via a template or menu. Once the user makes the choice, the system can continue. Other programming environments like COPE [3], POE [32], and MAGPIE [19, 89] use an intelligent parser which also uses a generator for error detection and repair.

Ambiguity is not a problem because the user uniquely defines the derivation tree. Going back to our expression example in Figure 12, we see that deciding whether $x \times y$ is multiplied first or $y + x$ is added first is totally determined by users who know what they wanted in the first place. Notice that this solution does not add anything to the grammar, unlike the solution proposed for acceptors.

Non-determinism is not a problem. When a choice has to be made, the users make the choice. They are presented options, and based on their own logic and the problem they are trying to solve, they make the correct choice the first time. Going back to the example with the if-then-else, there is no question which production to use because the users know whether they want the if-then statement or the if-then-else statement. Notice again that an ad hoc solution, like looking ahead or back tracking is not needed.

With a generative system the desires of users are recorded in the program that they are developing as opposed to being discerned as in the acceptor or translator system. Because of this capability, the generative systems are more powerful and useful than acceptor or translator systems.

4.4. Attributed Grammars

Context-free grammars were found not to be powerful enough to handle all aspects of any of the popular programming languages. There were semantic issues that had to be addressed which the context-free language could not. Included in these issues are such things as

- the scope rules of variables,
- the machine dependent constraints on the language like the size of an integer, and
- the context-sensitive issue of variables being declared before they are used.

Knuth [58, 59] introduced a more powerful grammar based on the context-free grammar that could handle some of these issues. This grammar was called an attributed grammar. It has since been proven that attributed grammars have the same power as a turing machine [109].

In this section we will define what attributes and semantic functions are. Then we will define what an attributed grammar is and give an example of it. We will look at some properties of the attributed grammar, and finally we will define what a time-varying attributed grammar is.

Attributes are place holders which are associated with symbols in a grammar and which take on values from different, possibly infinite sets. Each attribute is of type either synthesized or inherited. Synthesized attributes of a symbol derive their values from attributes of symbols that

are immediate descendants of the symbol in some derivation tree permitting information to flow up the tree. Inherited attributes of a grammar symbol derive their values from direct ancestors and sibling nodes in some derivation tree permitting information to flow down the tree.

Semantic functions determine how information flows up and down the derivation tree. Each function is used to define the computation of an attribute value of a symbol in the production in terms of other attribute values of symbols in the same production. The function for an attribute of the left-hand side symbol of a production computes a value for a synthesized attribute, and a function for an attribute of a symbol on the right-side of a production computes a value for an inherited attribute.

A specification of attributes A_G for a grammar $G = (v, \theta, P, \sigma)$ is a quintuple (S, I, Δ, F, M) where

- S is a finite set of synthesized attribute symbols disjoint from v but still a subset of V .
- I is a finite set of inherited attribute symbols disjoint from both v and S .
- Δ is a collection of domain sets of allowable attribute values.
- $F = \{f: \times_{k=0}^n d_k \rightarrow d_{n+1} \mid n \geq 0\}$ is a collection of functions defined over domain sets in Δ .
- M is a mapping from $S \cup I$ to Δ and is called the range function.

An attribute associator for a grammar G and a specification of attributes A_G is a mapping A from v to $2^{S \cup I}$ such that $S \in (V - v)$ and $I \in (V - v)$. The set $A(Y)$ is called the set of attributes associated with Y . For every Y in v , the set $A(Y) \cap S$, denoted by $S(Y)$, is the set of synthesized attributes associated with Y and the set $A(Y) \cap I$, denoted by $I(Y)$, is the set of inherited attributes associated with Y .

So that we can properly identify the attributes with their occurrence in productions, let us redefine production rules to have the following format:

$$P_i : C_{(i, 0)} ::= B_{(i, 1)} B_{(i, 2)} \dots B_{(i, n(i))}$$

where $n(i)$ is the number of symbols on the right hand side of production P_i .

We say that a production rule P_i has an attribute occurrence (a, j) if $0 \leq n(i)$ and $a \in A(B_{(i, j)})$. The set of attribute occurrences of P_i will be denoted by $AO(i)$. A semantic function for $\rho^i(a, j)$ for attribute occurrence (a, j) in production P_i is an $(i + 2)$ -tuple

$$\rho^i(a, j) = ((a, j), f^i(a, j), (b_1, k_1), (b_2, k_2), \dots, (b_m, k_m))$$

where

1. $(a, j) \notin AO(i)$,
2. $(b_l, k_l) \notin AO(i)$, for each l , $1 \leq l \leq m$,
3. $f^i(a, j) \in F$ is an m -ary semantic function for attribute a with $m \geq 0$, and
4. $\text{domain}(f) = \times_{l=1}^m M(b_l)$.

The value of (a, j) is to be evaluated using $f^i(a, j)$ and the values of

(b_i, k_i) . The set of (b_i, k_i) is often called the dependency set $D^i_{(a, j)}$ for the attribute occurrence (a, j) . A set of semantic functions for a production P_i is said to be valid if and only if it has one and only one rule to evaluate every attribute occurrence in the production.

We are now in a position to formally define an attributed grammar. An attributed grammar is a quadruple $G = (G_O, A_G, A, \text{sem})$ where

- $G_O = (v, \theta, P, \sigma)$ is a context-free grammar,
- A_G is a specification of attributes,
- A is an attribute associator for G and A_G , and
- sem is a semantic function associator for productions in G_O such that $\text{sem}(i)$ is a valid collection of semantic functions for production P_i in P .

As an example let us suppose we are defining a machine-dependent dialect of Pascal for use on a computer with 18-bit word and that, as a consequence, an unsigned integer constant is to be considered syntactically invalid if its value exceeds $2^{17} - 1$ ($= 131,071$). While the set of all unsigned numerals can easily be defined by

$G = (v, \theta, P, \sigma)$ where

$v = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \text{digit. numeral}\}$

$\theta = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$(v - \theta) = \{\text{digit. numeral}\}$

$P = \{P_1 : \langle \text{numeral} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{numeral} \rangle \langle \text{digit} \rangle$

$$P_2 : \langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \}$$

$$\sigma = \{\text{numeral}\},$$

the set of numerals $\{0, 1, \dots, 131070, 131071\}$ is difficult to define without an attributed grammar. Let us use an attributed grammar with attributes Val and Condition. Val corresponds to the domain of integers and is associated with both symbols $\langle \text{numeral} \rangle$ and $\langle \text{digit} \rangle$. Our attributed grammar is then defined by

$$G = (G_0, A_G, A, \text{sem}) \text{ where}$$

$$G_0 = (v, \theta, P, \sigma) \text{ where}$$

$$v = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \text{digit}, \text{numeral}\}$$

$$\theta = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$(v - \theta) = \{\text{digit}, \text{numeral}\}$$

$$P = \{P_1 : \langle \text{numeral} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{numeral} \rangle \langle \text{digit} \rangle$$

$$P_2 : \langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \}$$

$$\sigma = \{\text{numeral}\}$$

$$A_G = (S, I, \Delta, F, M) \text{ where}$$

$$S = \{\text{Val}, \text{Condition}\}$$

$$I = \{\emptyset\}$$

$$\Delta = \{\text{true}, \text{false}, 0, 1, 2, \dots\}$$

$$F = \{\text{Val}(\text{new}) = \text{value}(\text{from terminal string})$$

$$\text{Val}(\text{new}) = \text{Val}(\text{old}) \text{ -- the copy rule}$$

$$\text{Val}(\text{new}) = 10 \times \text{Val}(\text{right most child})$$

$$+ \text{Val}(\text{left most child})$$

$$\text{Condition} = \text{true if } \text{Val}(\text{new}) < 131,071$$

$$\text{Condition} = \text{false if } \text{Val}(\text{new}) \geq 131,071 \}$$

$$\mathbf{M} = \{\text{Val} \in \{0, 1, 2, \dots\}$$

$$\text{Condition} \in \{\text{true}, \text{false}\} \}$$

$$\mathbf{A} = \{\mathbf{S}(\mathbf{Y}), \mathbf{I}(\mathbf{Y})\} \text{ where}$$

$$\mathbf{S}(\langle \text{numeral} \rangle) = \{\text{Val}, \text{Condition}\}$$

$$\mathbf{S}(\langle \text{digit} \rangle) = \{\text{Val}\}$$

$$\mathbf{I}(\langle \text{numeral} \rangle) = \{\emptyset\}$$

$$\mathbf{I}(\langle \text{digit} \rangle) = \{\emptyset\}$$

$$\text{sem} = \{\text{sem}(1), \text{sem}(2)\} \text{ where}$$

$$\text{sem}(1) = \mathbf{P}_1 : \langle \text{numeral} \rangle ::= \langle \text{digit} \rangle$$

$$\text{Val}(\langle \text{numeral} \rangle) \leftarrow \text{Val}(\langle \text{digit} \rangle)$$

$$|\langle \text{numeral} \rangle|_2 \langle \text{digit} \rangle$$

$$\text{Val}(\langle \text{numeral} \rangle) \leftarrow 10 \times \text{Val}(\langle \text{numeral} \rangle_2)$$

$$+ \text{Val}(\langle \text{digit} \rangle)$$

$$\text{Condition} \leftarrow \text{Val}(\langle \text{numeral} \rangle) \leq 131,071$$

$$\text{sem}(2) = \mathbf{P}_2 : \langle \text{digit} \rangle ::= 0$$

$$\text{Val}(\langle \text{digit} \rangle) \leftarrow 0$$

...

...

9

Val($\langle \text{digit} \rangle$) \leftarrow 9

A derivation for an attributed grammar is the same as a derivation for an ordinary context-free grammar. A derivation tree for an attributed grammar $G = (G_O, A_G, A, \text{sem})$ is a derivation tree for G_O , in which the nodes are labeled with (x, τ) where $x \in v$ and τ is a mapping from $A(x)$ to $(\Lambda \cup (\cup_{a \notin A(x)} M(a)))$ (Λ is the undefined function) for each $a \notin A(x)$ and each $\tau(a) \notin \{\Lambda \cup M(a)\}$. In other words, (x, τ) is an assignment of values to attributes of node x in the derivation tree for an attributed grammar. This assignment of values is said to be valid if attributes associated with the nodes are either undefined (Λ) or have values consistent with the semantic functions of the grammar. That is, for each node (X_j, τ) and $a \notin A(X_j)$, either $\tau((a, j)) = \Lambda$, or

$\tau((a, j)) = f^i(a, j)((b_1, k_1), (b_2, k_2), \dots, (b_m, k_m))$ where

$((a, j), (b_1, k_1), (b_2, k_2), \dots, (b_m, k_m))$

is a valid semantic function for (a, j) in $\text{sem}(i)$.

In a derivation tree, an attribute instance (a, j) is said to be available if it is not undefined. A semantic function $\rho^i(a, j)$ is said to be applicable if all the attribute instances in $D^i_{(a, j)}$ are available or defined.

Let us continue with our example of the machine-dependent dialect of Pascal. We will use the string 673 for our example. Substituting N for numeral and D for digit, a derivation of 673 is

$N \Rightarrow ND \Rightarrow NDD \Rightarrow DDD \Rightarrow 6DD \Rightarrow 67D \Rightarrow 673$

The derivation tree and the values of the attributes Val and Condition are shown in figure 13.

AD-A158 102

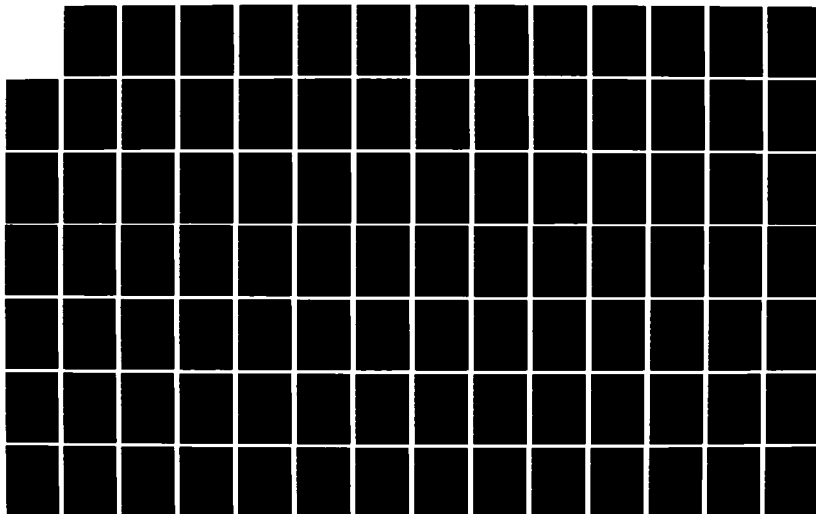
A META SYSTEM FOR GENERATING SOFTWARE ENGINEERING
ENVIRONMENTS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON
AFB OH W L MCKNIGHT 1985 AFIT/CI/NR-85-71D

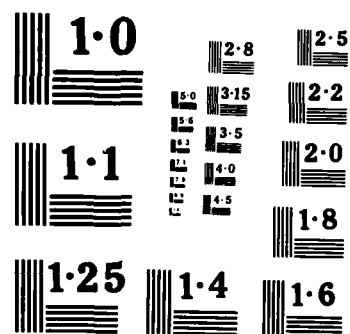
2/4

UNCLASSIFIED

F/G 9/2

NL





NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

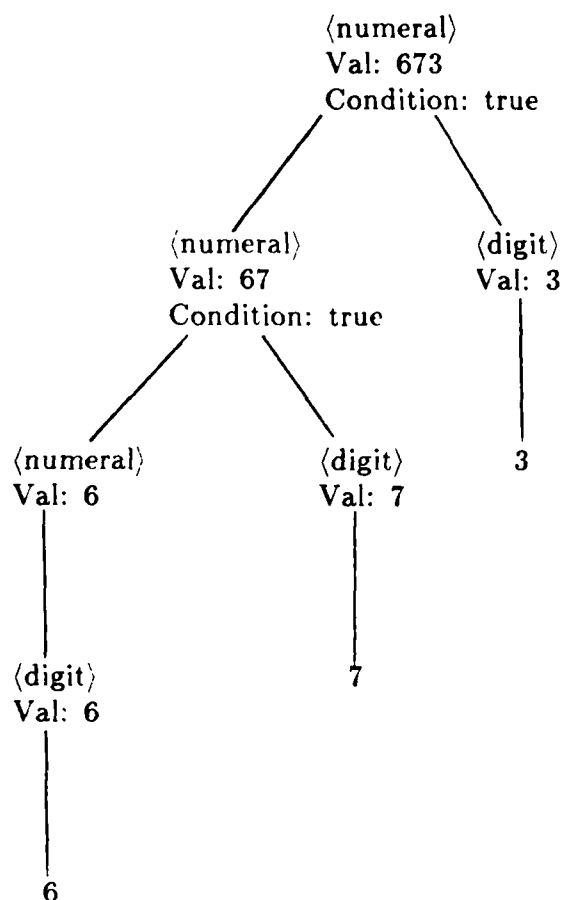


Figure 13: Derivation Tree of 673

The evaluated semantic tree for a derivation tree T of G is the tree obtained from T by the following algorithm, which is called the defining evaluator.

repeat

choose an applicable attribute instance (a, j) in T ;

evaluate the value of (a, j) ;

until no choice of (a, j) is possible;

Several evaluators have been proposed and analyzed in the literature [8, 10, 15, 20, 27, 28, 29, 35, 50, 52, 55, 56, 80, 84].

A grammar G is said to be locally circular if there exist two attribute occurrences (a, j) and (b, k) in a production such that the evaluation of one is dependent on the availability of the other. In other words, $(a, j) \in D^i_{(b, k)}$ and $(b, k) \in D^i_{(a, j)}$. Bochmann [10] has shown that, for a grammar which is known to be locally non-circular, it is easy to find, for each production P_i , an equivalent set of semantic functions which uses only the inherited attributes of the left symbol and the synthesized attributes of the symbols on the right hand side of the production. They satisfy the property

$$D^i_{(a, j)} \subset I(A) \cup (\cup_{j=1}^n S(B_j))$$

for $k = 0$ and $a \in S(X_0)$ as well as $k = 1, \dots, i$ and $a \in I(X_k)$.

An attribute grammar G is well-formed or non-circular if there does not exist any derivation tree T , such that the evaluated semantic tree for T would have an attribute instance (a, j) whose value is undefined. A grammar that does have an evaluated semantic tree with attribute instances whose values are undefined are said to be malformed or circular. Note that a grammar can be locally non-circular and still be globally circular. It is decidable whether any attribute grammar G is circular [58]. The circularity problem is also known to be NP-complete [51].

Going back again to our attributed grammar for the machine-dependent dialect of Pascal, we can see that our attributed grammar is well formed. The dependency graph shown in figure 14 is acyclic. Notice also that the derivation tree shown in figure 13 does not have an attribute instance with an undefined value.

A type of attribute called the time varying attribute was introduced by

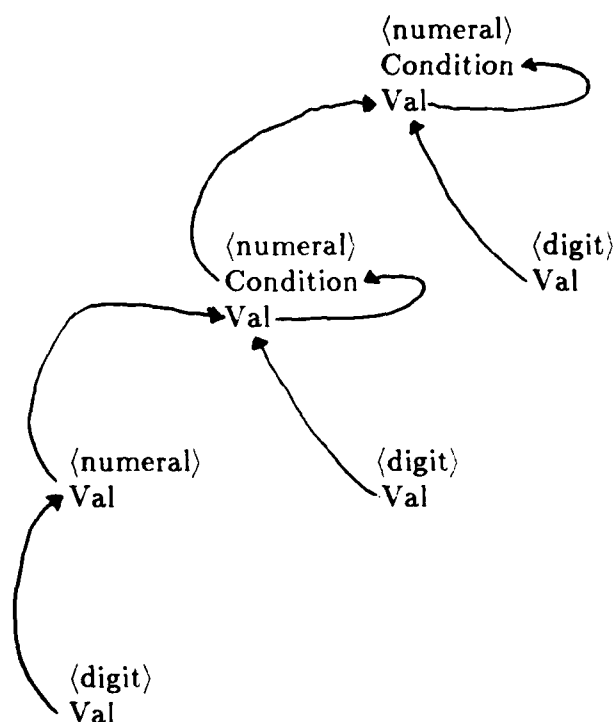


Figure 14: Dependency Graph for Machine-Dependent Pascal

Skedzeleski [93]. Although the idea is quite powerful and gets rid of some of the problems with regular attributes, the idea has not really taken hold in the computer field. A time-varying attributed grammar is an attributed grammar in which each attribute occurrence may be given an initial value that can be changed during the course of the evaluation. The evaluator for time-varying attributes is more complicated than previous evaluators since it must schedule both initial and iterative evaluations of attributes. It must also be able to remember attribute dependencies after the initial evaluation has taken place so that changes in attribute values will propagate around the parse tree in accordance with the attribute definitions. An example of the specification of a time-varying attributed grammar is given by a time-varying attributed grammar which includes the following production with its associated semantic functions:

$P_1: \langle S \rangle ::= \langle A \rangle \langle B \rangle$

$\text{Span}(\langle S \rangle) \leftarrow \text{Len}(\langle A \rangle) + \text{Len}(\langle B \rangle)$

$\text{Len}(\langle A \rangle) \leftarrow 15$

if $\text{Len}(\langle A \rangle) > 10$ then $\text{Len}(\langle B \rangle) \leftarrow 2$

Let us suppose that the only attribute that symbol $\langle S \rangle$ has is a synthesized attribute called Span, the only attribute that symbol $\langle A \rangle$ has is a synthesized attribute called Len, and the only attribute that symbol $\langle B \rangle$ has is an inherited attribute called Len which is initially set to 1. In the reevaluation process, the attribute $\text{Len}(\langle B \rangle)$ will be changed to 2 and the attribute $\text{Span}(\langle S \rangle)$ will be set to 17.

In a regular attributed grammar, circularity was a problem basically because of the undefined state of an attribute. In time-varying attributed grammar this is not a problem because all attributes can have an initial value. The hazards of using time-varying attributes include:

1. Non-terminating algorithms can be specified.
2. Results may depend on the initial values of the attributes.
3. Results may depend on the evaluation order of the semantic rules.

These problems already exist in regular attributed grammars. The first two problems were hidden in the functions used in attribute definitions; the last problem was implicit in the evaluation mechanisms used.

4.5. Attributed Grammars with Right Regular Parts

A special case of the attributed grammar is the attributed grammar with right regular parts. In this section we will look at what changes need to be made to the semantics of a grammar so that the attributed grammar and the attributed grammar with right regular parts can be equivalent. For a more detailed discussion of this topic, see [9].

Normally, the grammar for an attributed grammar is specified in BNF. This is the format that we have been using in this presentation. However, extensions to include regular expressions have been used for the syntactic specification of programming languages with the advantage of better readability. This is best illustrated with an example.

Let us consider a typical programming language construct of an arithmetic expression which consists of a sequence of terms separated by addition operators:

$$\langle \text{expr} \rangle \leftarrow \langle \text{term} \rangle + \langle \text{term} \rangle + \dots + \langle \text{term} \rangle$$

Let us associate some semantics with this expression. Let us first define the type of the expression as being real if any term of the expression is real, otherwise it is integer. This can be accomplished if we formulate the following semantic rule: The $\langle \text{expr} \rangle$ is real, if the $\langle \text{expr} \rangle$ contains a $\langle \text{term} \rangle$ which is real, otherwise it is integer.

Notice that the order in which the terms are evaluated has no bearing on the type of expression. Next let us define the arithmetic value of the expression. Again, let us associate an attribute with the expression and the terms which represent their arithmetic values. We define the value of the expression by a successive addition of the values of the terms using the concept of a variable which holds the intermediate results and

has the initial value zero. Notice this time the order in which the terms are considered is important if arithmetic overflow is possible. In BNF notation we would have the following:

$$\langle \text{expr} \rangle ::= \langle \text{term list} \rangle$$

$$\text{value}(\langle \text{exp} \rangle) \leftarrow \text{value}(\langle \text{term list} \rangle)$$

$$\text{type}(\langle \text{exp} \rangle) \leftarrow \text{type}(\langle \text{term list} \rangle)$$

$$\langle \text{term list} \rangle_0 ::= \langle \text{term} \rangle + \langle \text{term list} \rangle_1$$

$$\text{value}(\langle \text{term list} \rangle_0) \leftarrow \text{value}(\langle \text{term} \rangle) + \text{value}(\langle \text{term list} \rangle_1)$$

$$\text{if } (\text{type}(\langle \text{term} \rangle) == \text{type}(\langle \text{term list} \rangle_1)) \text{ then}$$

$$\text{type}(\langle \text{term list} \rangle_0) \leftarrow \text{type}(\langle \text{term} \rangle)$$

$$\text{else}$$

$$\text{type}(\langle \text{term list} \rangle_0) \leftarrow \text{real}$$

$$| \langle \text{term} \rangle$$

$$\text{value}(\langle \text{term list} \rangle_0) \leftarrow \text{value}(\langle \text{term} \rangle)$$

$$\text{type}(\langle \text{term list} \rangle_0) \leftarrow \text{type}(\langle \text{term} \rangle)$$

For our particular example, the production rule of this expression would be

$$\langle \text{exp} \rangle ::= \langle \text{term} \rangle [+ \langle \text{term} \rangle]^*$$

In order to be able to do attribute evaluation on regular expressions some modifications need to be made to the attribute evaluation mechanisms:

1. With each occurrence of a repeated subexpression, α^* , within

the right side of a production rule, a set of so called local attributes are associated with the production. Each local attribute represents an intermediate result which is reevaluated for each occurrence of α within the derivation tree.

2. For each local attribute of a repeated subexpression, α^* , there is
 - a. a semantic function for initialization, which specifies the initial attribute value as a function of the values of other attributes, except those which are associated with α or subexpressions of α .
 - b. a semantic function for redefinition, which specifies a new attribute value as a function of previous local attribute values of α , and values of attributes which are associated with α or subexpressions of α .
3. An order is specified in which the different repetitions of α within the derivation tree are considered for semantic evaluation (redefinition of local attributes).
4. The values of the local attributes of α^* (before redefinition) can be used by semantic functions for the evaluation of attributes associated with α or subexpressions of α .
5. The final values of the local attributes of α^* can be used by semantic functions for the evaluation of attributes, except those which are associated with α or subexpressions of α .

Using our example again, let us see how this mechanism can be used

to describe the evaluation of the type and the value of expression. This can be done as follows:

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle_0 [+ \langle \text{term} \rangle_1]^*$

(initializing local attributes)

$\text{interm-type} \leftarrow \text{type} (\langle \text{term} \rangle_0)$

$\text{interm-value} \leftarrow \text{value} (\langle \text{term} \rangle_0)$

(redefining local attributes)

if $(\text{interm-type} \neq \text{type} (\langle \text{term} \rangle_1))$ then

$\text{interm-type} \leftarrow \text{real}$

$\text{interm-value} \leftarrow \text{interm-value} + \text{value} (\langle \text{term} \rangle_1)$

(defining non-local attributes)

$\text{type} (\langle \text{exp} \rangle) \leftarrow \text{interm-type}$

$\text{value} (\langle \text{exp} \rangle) \leftarrow \text{interm-value}$

(order for considering the repetition of $[+ \langle \text{term} \rangle_1]$ for

semantic evaluation)

from left to right

The formalism of attributed grammars with right regular parts is more complex than those of attributed grammars. However, the power of both are the same. The biggest merit of the attributed grammar with right regular parts is that it is more comprehensible and concise. We also observe that the depth of the derivation tree can be considerably reduced with attributed grammars with right regular parts. Another advantage of these grammars is that regular expressions represent an alternative to

left recursion and can thus be used in top-down syntax analysis. Also, because the relative order of evaluation for the different repetitions of a subexpression must be specified explicitly, the type of attribute evaluation could be picked to lessen the number of passes needed by the attribute evaluator.

4.6. Grammar Forms

Although grammar forms have been around for sometime, our use as models for systems is new. In this section we will informally introduce what we mean by grammar forms. We will then build up the terminology needed to formally define grammar forms. We will discuss some techniques used in grammar forms. We will then define the grammar form needed to define the class of context-free grammars in Chomsky Normal Form. We will extend our grammar forms to allow context-free grammars with right regular parts and show what problems we would have had.

In order for grammar forms to have any meaning at all, we first need to introduce the notion of grammar similarity or equivalence. This will lead us into the notions of families of grammars.

Intuitively, we would think that two grammars were similar or equivalent if the language generated by those two grammars were equal. Let us use an example to illustrate what we mean. For simplicity, when we define a grammar, we will only list the production rules where it is apparent what the vocabulary, terminal symbols, non-terminal symbol, and start symbols are from the production rules themselves. Let G_1 be defined as

$$S ::= S01$$

$$S ::= 1$$

and G_2 be defined as

$$T ::= T0T$$

$$T ::= 1$$

Clearly, both grammars generate the language $L = 1(01)^*$. However, G_2 is ambiguous, whereas G_1 is not. We do not wish to consider G_1 and G_2 to be structurally equivalent, because G_2 associates many distinct derivation trees with sentences of L , whereas G_1 provides a unique derivation tree for each sentence. We will however, define this type of equivalence as a weak equivalence. Therefore, given any two grammars G_1 and G_2 , we say they are weakly equivalent if $L(G_1) = L(G_2)$.

It appears then, the only way we can have equivalence or strong equivalence, is to require that if sentences in one grammar are ambiguous, they must also be in the other grammar. This can be met by requiring that the derivations of the two grammars have a one-to-one correspondence for each sentence generated. Yet this criterion may be too strict, for one form of structural ambiguity is trivial and easy to remove. For example let grammar G_1 be defined as

$$S ::= S01$$

$$S ::= 1$$

and grammar G_2 be defined as

$$S ::= S$$

$$S ::= S01$$

$$S ::= 1$$

Grammar G_2 has all the productions of G_1 plus the production rule S

$::= S$. Since this rule may be applied an arbitrary number of times to any sentential form containing S , any string generated by G_2 has an infinity of derivations. In contrast, G_1 generates each string by a unique derivation. Although G_2 is very ambiguous, the only difference between the structural descriptions of any sentence according to the two grammars is the number of repetitions of certain sentential forms in each derivation. Therefore, we will define a minimal derivation to be a derivation in which no sentential form is repeated in the derivation.

With the definition of minimal derivation, we can now define what we mean by strong equivalence. Let G_1 and G_2 be two grammars. They are said to be strongly equivalent, if they are weakly equivalent and, for each terminal string ω , the minimal derivation of ω permitted by G_1 can be replaced in a one-to-one correspondence with those permitted by G_2 .

Another concept that needs to be introduced is the notion of grammar morphism. Let $G_1 = (v_1, \theta_1, P_1, \sigma_1)$ and $G_2 = (v_2, \theta_2, P_2, \sigma_2)$ be two grammars and $h: v_1^* \rightarrow v_2^*$ be a homomorphism such that:

1. $h(v_1 - \theta_1) \subseteq v_2 - \theta_2$.
2. $h(\theta_1) \subseteq \theta_2^*$.
3. for all $A_1 \Rightarrow \alpha_1$ in P_1 , $h(A_1 - \alpha_1) = A_2 \Rightarrow^+ \alpha_2$ in G_2
where $h(A_1) = A_2$ and $h(\alpha_1) = \alpha_2$, and
4. $h(\sigma_1) = \sigma_2$.

We say that h is a grammar morphism because h maps G_1 to G_2 .

Grammar morphisms are the most systematic notions of grammar

equivalence. The grammar form concepts are based on two special kinds of grammar morphisms. We say that a grammar morphism $h: G_1 \rightarrow G_2$, where $G_1 = (v_1, \theta_1, P_1, \sigma_1)$ and $G_2 = (v_2, \theta_2, P_2, \sigma_2)$ is

1. fine if $h(A_1 \Rightarrow \alpha_1)$ is in P_2 for all $A_1 \Rightarrow \alpha_1$ in P_1 ,
2. length preserving if $h(v_1) \subseteq v_2$, and
3. very fine if h is fine and length preserving.

The notion of grammar morphisms are going to help us in determining grammar equivalence. As we shall see latter, whether one grammar is a strict interpretation of another is equivalent to whether there exists a very fine grammar morphism between them.

Informally, a grammar form is a grammar that can be used to generate a family of grammars. Let G be a grammar form; then for each grammar G_i in G , each production rule in G_i is an image of some production rule in G . If terminals are ignored, there is a very fine grammar morphism from G_i to G . Each G_i in G can be systematically derived by a process of substitutions called interpretation. For restricted versions of interpretations, the question of equivalence between two grammars is decidable. We will use a restricted version of interpretation to model tuning.

Two types of interpretations have been studied in the literature, the g- (or general) interpretation [7] first introduced by Ginsburg, and s- (strict) interpretation [71] first introduced by Salomaa. The s-interpretation is more general than the g-interpretation and since we are interested in representing trees and not languages, we will work mostly with the s-interpretation.

A context-free grammar $G = (V, \theta, P, \sigma)$ is in standard form if each production has one of the following formats:

$A ::= aB$ or $A ::= a$ where

$A \in V - \theta$

$B \in V^*$ and

$a \in \theta$

This canonical form is called Greibach normal form.

Theorem 11: Every context-free language without λ can be constructed by a strongly equivalent grammar in Greibach normal form.

See [45] for the proof of this theorem.

Now let take a couple of examples of grammar forms and see what classes of context-free grammars can be generated from them. First, let us use a grammar form G_1 where it has two productions, namely,

$S ::= SS$ and

$S ::= a$

Any s-interpretation on G_1 will give us a grammar in Chomsky normal form, and thus G_1 can be used to generate all context-free grammars.

For our second example, let us use a grammar form G_2 where it has two productions, namely,

$S ::= aS$ and

$S ::= a$

Any s-interpretation on G_2 will give us a grammar in Greibach normal form, and thus G_2 can be used to generate all context-free grammars.

4.6.1. Normal Form

One of advantages of using grammar forms is that so many classes of context-free grammars can be generated from a very few grammar forms. In order to illustrate this we first need to discuss context-free grammar canonical forms. There are two canonical forms that are used and studied quite frequently in programming language studies, namely normal form and standard form. Canonical forms for context-free grammars are useful for two reasons:

1. it is often easier to establish some property for canonical-form grammars than for context-free grammars in general, and
2. representing context-free languages by canonical-form grammars may make it easier to devise methods of syntax analysis for the languages.

A context-free grammar $G = (V, \theta, P, \sigma)$ is in normal form if each production has one of the following formats:

$A ::= BC$ or $A ::= a$ where

$A, B, C, \in V - \theta$ and

$a \in \theta$

This canonical form is also called Chomsky normal form.

Theorem 10: Any context-free language without λ can be constructed by a strongly equivalent grammar in Chomsky normal form.

See [45] for the proof of this theorem.

$$\theta = \{a, b, c, d\}$$

$$P = \{P_1 : T ::= S$$

$$P_2 : T ::= R$$

$$P_3 : S ::= aSb$$

$$P_4 : S ::= ab$$

$$P_5 : R ::= cRd$$

$$P_6 : R ::= cd \}$$

$$\sigma = \{T\}$$

Our final technique is the intersection of two grammar forms. Given two grammar forms G_1 and G_2 , one can construct a grammar form G such that $G_s(G) \supseteq G_s(G_1) \cap G_s(G_2)$ and $G_{hs}(G) \supseteq G_{hs}(G_1) \cap G_{hs}(G_2)$ as follows:

$$v = ((v_1 - \theta_1) \times (v_2 - \theta_2) \cup (\theta_1 \times \theta_2),$$

$$\theta = (\theta_1 \times \theta_2),$$

$$\sigma = [\sigma_1, \sigma_2], \text{ and}$$

P consists of the following productions:

for each $P_{1,i} : A ::= X_1 \dots X_n$ in P_1 and

for each $P_{2,i} : B ::= Y_1 \dots Y_n$ in P_2 such that

for each $[X_i, Y_i]$ in v , $1 \leq i \leq n$,

$[A, B] \supseteq [X_1, Y_1] \dots [X_n, Y_n]$ is in P .

Furthermore, $G \subseteq_s G_1$ and $G \subseteq_s G_2$.

$$P = \{P_1 : S ::= RSb$$

$$P_2 : S ::= Rb$$

$$P_3 : R ::= cRd$$

$$P_4 : R ::= cd \}$$

$$\sigma = \{S\}$$

The technique of union combines the alphabets and productions of two grammar forms. Let $G_1 = (v_1, \theta_1, P_1, \sigma_1)$ and $G_2 = (v_2, \theta_2, P_2, \sigma_2)$ be two grammar forms. Also, let $G = (v, \theta, P, \sigma)$ be the union of G_1 and G_2 where

$$v = \{v_1 \cup v_2 \cup \{\sigma\}\},$$

$$\theta = \{\theta_1 \cup \theta_2\}, \text{ and}$$

$$P = \{P_1 \cup P_2 \cup \{\sigma ::= \sigma_1, \sigma ::= \sigma_2\}\}.$$

Clearly if G_1 and G_2 have disjoint alphabets, $G_s(G) = G_s(G_1) \cup G_s(G_2)$ and $G_{hs}(G) = G_{hs}(G_1) \cup G_{hs}(G_2)$. If the two alphabets are not disjoint, then all we can prove is that $G_s(G) \supseteq G_s(G_1) \cup G_s(G_2)$ and $G_{hs}(G) \supseteq G_{hs}(G_1) \cup G_{hs}(G_2)$.

Let us illustrate the technique of union by an example. Let G_1 be defined with two production rules ($S ::= aSb$ and $S ::= ab$) and G_2 also be defined with two production rules ($R ::= cRd$ and $R ::= cd$) where $\{a, b\}$ and $\{c, d\}$ are the terminal symbols of the respective grammar forms. If we do the union, we need to introduce a new non-terminal which we will let be $\{T\}$. Then our new grammar would be $G = (v, \theta, P, \sigma)$ where,

$$v = \{b, c, d, R, S, T\}$$

The replacement technique is a generalization of isolation. This technique replaces a subgrammar form of the grammar form by its interpretation grammar form. Clearly, the resultant grammar form is an interpretation grammar form of the original.

The technique of substitution combines two grammar forms G_1 and G_2 in such a way that a leaf node representing a terminal in G_1 is substituted by a tree derived by G_2 . Given $G_1 = (v_1, \theta_1, P_1, \sigma_1)$ and $G_2 = (v_2, \theta_2, P_2, \sigma_2)$ with disjoint alphabets. Let the new grammar form be $G = (v, \theta, P, \sigma)$ where

- $v = \{(v_1 - \{a\}) \cup v_2\},$
- $\theta = \{(\theta_1 - \{a\}) \cup \theta_2\},$
- $P = \{\tau(P_1) \cup P_2\}$ where a is in θ and $\tau(P_1)$ is in P_1 in which a is replaced by σ_2 , and
- $\sigma = \sigma_1.$

Intuitively a primitive in G_1 is further expanded by G_2 .

Let us illustrate the technique of substitution by an example. Let G_1 be defined with two production rules ($S ::= aSb$ and $S ::= ab$) and G_2 also be defined with two production rules ($R ::= cRd$ and $R ::= cd$) where $\{a, b\}$ and $\{c, d\}$ are the terminal symbols of the respective grammar forms. If we make the substitution of the terminal $\{a\}$ with the tree generated with G_2 , then it would be the same as if we have defined $G = (v, \theta, P, \sigma)$ as follows:

$$v = \{b, c, d, R, S\}$$

$$\theta = \{b, c, d\}$$

There are various techniques for manipulating grammar forms in order to generate grammar forms like we have just discussed. The techniques we will look at are isolation, replacement, substitution, union, and intersection. We will begin with isolation.

Sometimes it may be desirable to isolate derivations from a non-terminal in the interpretation grammar form. We want to restrict the derivation trees that may be generated by a non-terminal. This is usually the case when experience has proven the merit of one derivation with respect to another. Let us consider a grammar form $G = (v, \theta, P, \sigma)$ and terminating derivation $A \Rightarrow^+ \alpha_i$ where $\alpha_i \in \theta^*$ for $1 \leq i \leq n$, for some $n \geq 1$ and some $A \in v - \theta$. We can derive either $G_1 \approx_s G$ or $G_1 \approx_{hs} G$, such that

1. $G_1 = (v_1, \theta, P_1, \sigma)$ with $v \subseteq v_1$,
2. all productions in P whose left hand side is not A are taken into P_1 unchanged, and
3. the remaining productions in P_1 only serve to derive the α_i from A and nothing else.

We then say the the derivations $A \Rightarrow^* \alpha_i$, $1 \leq i \leq n$ in G have been isolated in G_1 . Notice that any derivation from A must always go through one of the α_i 's. G_1 can be constructed by using extra non-terminals which are used in only one production. Since we have already shown that we can restrict derivations from a non-terminal by either an s-interpretation or hs-interpretation, we will avoid generating this grammar form all together and use other means to restrict the derivations.

Theorem 8: Let $G_1 = (v_1, \theta_1, P_1, \sigma_1)$ and $G_2 = (v_2, \theta_2, P_2, \sigma_2)$ be two grammars. Then

1. $G_2 \approx_s G_1(\mu)$ iff $\mu^{-1}: G_2 \rightarrow G_1$ is a very fine grammar morphism, and
2. $G_2 \approx_{hs} G_1(\mu)$ iff $\mu^{-1}: G_2 \rightarrow G_1$ is a length preserving grammar morphism.

See [109] for the proof of this theorem.

It can clearly be seen that two systematic techniques of generating similar grammars are tied to the concept of grammar morphisms preserving structural properties. There is also a relationship between the derivation of the s- and hs-interpretation grammar and derivations of its form grammar.

Theorem 9: Let $G_1 = (v_1, \theta_1, P_1, \sigma_1)$ and $G_2 = (v_2, \theta_2, P_2, \sigma_2)$ be two grammars such that $G_2 \approx_{hs} G_1$ (and because of preserving properties $G_2 \approx_s G_1$). Then for every derivation $\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_n$ in G_2 for some β_i in v_2^* , $0 \leq i \leq n$ and $n > 0$, there is a derivation $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_m$ such that β_i is in $\mu(\alpha_i)$ for $0 \leq i \leq m$ and $n = m$. When $G_2 \approx_{hs} G_1$, then $\alpha_i = \mu^{-1}(\beta_i)$, $0 \leq i \leq m$.

See [109] for the proof of this theorem.

This theorem states that for each derivation tree τ_2 in G_2 there is exactly one derivation tree τ_1 in G_1 . Clearly, τ_1 and τ_2 can be obtained from each other by simply relabeling their nodes.

hs-grammar family of a grammar form G , denoted by $G_{hs}(G)$, properly includes the s-grammar family of G , while the hs-grammatical family of G , denoted by $L_{hs}(G)$, is the same as the s-grammatical family of G .

Let $G_1 = (v_1, \theta_1, P_1, \sigma_1)$ and $G_2 = (v_2, \theta_2, P_2, \sigma_2)$ be two grammars. G_2 is called an hs-interpretation grammar modulo μ of G_1 if there exists a dfl-substitution μ on v_1^* such that

1. $\mu(A) \subseteq v_2 - \theta_2$, for all A in $v_1 - \theta_1$,
2. $\mu(a) \subseteq \theta_2$, for all a in θ_1 ,
3. $P_2 \subseteq \mu(D(G_1))$, where $\mu(D(G_1)) = \{\mu(A \Rightarrow^* a) : A \Rightarrow^* a \text{ in } D(G_1)\}$, where $D(G_1)$ is the set of derivations of G_1 and $\mu(A \Rightarrow^* a) = \mu(A) \Rightarrow^* \mu(a)$, and
4. σ_2 is in $\mu(\sigma_1)$,

and is denoted by $G_2 \approx_{hs} G_1(\mu)$.

The hs-interpretation is similar to the s-interpretation except that the productions of the target grammar are images of source grammar derivations. The hs-grammar family and hs-grammatical family can also be defined analogous to s-grammar family and the s-grammatical family. Likewise, we can define hs-form equivalent and strong hs-form equivalent as having the same properties as s-form equivalent and strong s-form equivalent. Also, the decidability problem and its NP-completeness can be proved similar to the s-interpretation proofs. Clearly, an s-interpretation is an hs-interpretation, therefore, for any grammar form G , then the following also holds, namely: $G_s(G) \subseteq G_{hs}(G)$ and $L_s(G) \subseteq L_{hs}(G)$. The hs-interpretation is needed in modeling the tuning process of a derivation tree and its underlying grammar.

Theorem 4: Let $G = (v, \theta, P, \sigma)$ be a grammar form. For every two grammar forms F_1 and F_2 which are s-interpretations of G and are production minimal, symbol tight and strongly s-form equivalent to G , F_1 and F_2 are isomorphic (or minimal).

See [109] for the proof of this theorem.

Theorem 5: \approx_s is decidable for grammar forms.

See [109] for the proof of this theorem.

Theorem 6: The decidability of \approx_s for grammar forms is an NP-complete problem.

See [109] for the proof of this theorem.

In fact, for grammar forms, we do not need to restrict ourselves to s-interpretations. Blattner has given us a more powerful theorem.

Theorem 7: The equivalence of context-free grammar forms is decidable.

See [7] for the proof of this theorem.

In general, the problem of deciding whether $L(G_1) = L(G_2)$ for arbitrary context-free grammars G_1 and G_2 is unsolvable [21, 45]. This is why a systematic procedure to generate similar grammars is superior to a procedure for checking if two arbitrary grammars are similar.

Before discussing different techniques for systematically modifying grammars, we will define a new type of interpretation called the homomorphic strict-interpretation (or hs-interpretation). This interpretation is more general than strict-interpretation in that the

$$P_2 = \{P_{2.1} : \langle C \rangle ::= \langle D \rangle \langle c \rangle$$

$$P_{2.2} : \langle D \rangle ::= \langle D \rangle \langle c \rangle$$

$$P_{2.3} : \langle D \rangle ::= \langle D \rangle \langle d \rangle$$

$$P_{2.4} : \langle D \rangle ::= \langle d \rangle \}$$

It should be fairly easy to see that G_2 generates all strings of the form $d(d \cup c)^*c$.

The collection of s-interpretation grammars derived from the grammar form G of the form grammar F is referred to as the s-grammar family of F and is denoted by $G_s(G)$. Similarly, the collection of languages obtained from G is referred to as the s-grammatical family of G and is denoted by $L_s(G)$. It is defined by $L_s(G) = \{L(G_i) : G_i \approx_s G\}$. As we had equivalence relationships for context-free grammars, likewise we can define similar equivalence relationships for grammar forms. Two grammar forms F_1 and F_2 are s-form equivalent if $L(F_1) = L(F_2)$ and strong s-form equivalent if $G_s(F_1) = G_s(F_2)$. Notice how the s-form equivalence corresponds to weak equivalence and strong s-form equivalence corresponds to strong equivalence.

In our earlier discussion of equivalence, we defined a minimal derivation in order to come up with our strong equivalence definition. Minimal derivation also allows us to define a minimal grammar for a language. The same concepts also apply to grammar forms. A grammar form G_1 is production minimal if there is no grammar form G_2 such that $G_1 \approx_s G_2$ and $G_2 \approx_s G_1$ and G_2 has fewer productions than G_1 . Similarly G_1 is symbol minimal if there is no grammar form G_2 such that $G_1 \approx_s G_2$ and $G_2 \approx_s G_1$ and G_2 has fewer symbols than G_1 . Finally, G_1 is symbol tight if G_1 has no useless symbols in its vocabulary.

$$P_{1.2} : \langle A \rangle ::= \langle a \rangle \}$$

$$\sigma_1 = \langle A \rangle$$

$$V_1 = \{c, d, C, D\}$$

$$_1 = \{c, d\}$$

ζ_1 is the μ_s interpretation that performs the following mapping

$$\mu_s(a) = \{c, d\}$$

$$\mu_s(A) = \{C, D\}$$

$$\mu_s(P_1) = \{P_{1.1} : \langle C \rangle ::= \langle C \rangle \langle c \rangle$$

$$P_{1.2} : \langle C \rangle ::= \langle C \rangle \langle d \rangle$$

$$P_{1.3} : \langle C \rangle ::= \langle c \rangle$$

$$P_{1.4} : \langle C \rangle ::= \langle D \rangle \langle c \rangle$$

$$P_{1.5} : \langle C \rangle ::= \langle D \rangle \langle d \rangle$$

$$P_{1.6} : \langle C \rangle ::= \langle d \rangle$$

$$P_{1.7} : \langle D \rangle ::= \langle C \rangle \langle c \rangle$$

$$P_{1.8} : \langle D \rangle ::= \langle C \rangle \langle d \rangle$$

$$P_{1.9} : \langle D \rangle ::= \langle c \rangle$$

$$P_{1.10} : \langle D \rangle ::= \langle D \rangle \langle c \rangle$$

$$P_{1.11} : \langle D \rangle ::= \langle D \rangle \langle d \rangle$$

$$P_{1.12} : \langle D \rangle ::= \langle d \rangle \}$$

$$\mu_s(\sigma) = \{C\}$$

Then we define another form grammar G_2 such that $G_2 \approx_s G_1$ where G_2 has the following set of productions:

- V is an infinite vocabulary,
- $\psi \in V$ is an infinite vocabulary of terminal symbols, and
- ζ is the allowable set of interpretations of G using the symbols in the two vocabulary sets.

F represents the family of grammars that are interpretations of the form grammar G and the process of interpretations is a systematic technique for generating them.

A grammar form $F_2 = (G_2, V_2, \psi_2, \zeta_2)$ is called s-interpretation grammar form of a grammar form $F_1 = (G_1, V_1, \psi_1, \zeta_1)$ if $G_2 \approx_s G_1$, $V_2 \rightarrow V_1$, $\psi_2 \rightarrow \psi_1$, and $\zeta_2 \rightarrow \zeta_1$.

As an example of a grammar form, let us use the family of grammars called the left linear grammars. By definition, left linear grammars are regular grammars which have productions of the following format:

$$\begin{aligned} A &\Rightarrow B\alpha \\ A &\Rightarrow \alpha \quad \text{where} \\ A &\in (V - \Theta) \cup \{\sigma\} \\ B &\in (V - \Theta) \\ \alpha &\in \Theta \end{aligned}$$

Let $F_1 = (G_1, V_1, \psi_1, \zeta_1)$ where

$G_1 = (v_1, \theta_1, P_1, \sigma_1)$ where

$$v_1 = \{A, a\},$$

$$\theta_1 = \{a\},$$

$$v_1 - \theta_1 = \{A\},$$

$$P_1 = \{P_{1,1} : (A ::= (A, a))\}$$

to be infinite, if that vocabulary is bounded we will have no problems. The productions are obtained by taking images of productions in the source grammar. The starting symbol is an image of σ_1 . The g-interpretation can also be similarly obtained.

Clearly, s-interpretations are length preserving while g-interpretations are not. Also, every s-interpretation is a g-interpretation while the converse is not true.

Whenever $P_2 = \mu(P_1)$, we say that G_2 is a full s-interpretation of G_1 if we are doing s-interpretations, or a full g-interpretation of G_1 if we are doing g-interpretations. Full s-interpretation is denoted by $G_2 \approx_{fs} G_1$. Full g-interpretation is denoted by $G_2 \approx_{fg} G_1$.

Let us illustrate an s-interpretation with a very simple form grammar. Let G_1 be a grammar with one production

$$P_{1.1} : S ::= a$$

and let G_2 be a grammar with two productions, namely

$$P_{2.1} : T ::= b$$

$$P_{2.2} : T ::= c$$

Let $\mu(S) = \{T\}$, $\mu(a) = \{b, c\}$, and $\mu(S \rightarrow a) = \{T \rightarrow b, T \rightarrow c\}$. Then we can say that $G_2 \approx_s G_1$. In fact, we can even say that $G_2 \approx_{fs} G_1$.

We are now in a position to formally define a grammar form. A grammar form F is a quadruple (G, V, ψ, ζ) where

- $G = (v, \theta, P, \sigma)$ is a context-free grammar, often referred to as the form grammar of F ,

Both interpretations use the notion of a disjoint-finite-letter substitution (df-l-substitution). Let U, V be two alphabets and μ be a letter substitution from U to 2^V . Then μ is a dfl-substitution if for all X, Y in U , $\mu(X) \cap \mu(Y) = \emptyset$ when $X \neq Y$.

Now let us define what a s-interpretation is. Let $G_1 = (v_1, \theta_1, P_1, \sigma_1)$ and $G_2 = (v_2, \theta_2, P_2, \sigma_2)$ be two grammars. G_2 is called an s-interpretation grammar modulo μ of G_1 (denoted by $G_2 \approx_s G_1$) if there exists a dfl-substitution μ on v_1^* such that

1. $\mu(A) \subseteq v_2 - \theta_2$, for all A in $v_1 - \theta_1$,
2. $\mu(a) \subseteq \theta_2$, for all a in θ_1 ,
3. $P_2 \subseteq \mu(P_1)$, where $\mu(P_1) = \{\mu(A \rightarrow a) : A \rightarrow a \text{ in } P_1\}$,
where $\mu(A \rightarrow a) = \mu(A) \rightarrow \mu(a)$, and
4. σ_2 is in $\mu(\sigma_1)$.

The substitution μ is referred to as an s-interpretation of G_1 . The substitution must be disjoint in order to preserve the structural properties of the grammar. The definitions of g-interpretation and s-interpretation grammars are similar except that terminals can be replaced by sets of terminal words rather than just by sets of terminal letters. For both interpretations, G_1 is known as the source, master or form grammar, and G_2 is the image, target or interpretation grammar.

The s-interpretation is obtained for the source grammar by mapping distinct terminals into disjoint sets of terminals and distinct non-terminals into disjoint sets of non-terminals. Although the vocabulary making up the set of terminals and the set of non-terminals is supposed

For our third example, let us use a grammar form G_3 where it has three productions, namely,

$$S ::= aSS$$

$$S ::= aS$$

$$S ::= a$$

Any s-interpretation on G_3 will give us a grammar in what is known as Greibach 2-standard normal form which is also known to be able to generate all context-free languages. Thus, G_3 can be used to generate all context-free grammars.

For our final example, let us use a grammar form G_4 where all the productions of the grammar form have the following format:

$$S ::= a^i S a^j S a^k \text{ where } i + j + k = 9$$

This grammar has the appearance of being in some type of normal form. Wood in [109] has proved that any grammar form in normal form will generate all context-free languages. Again, any s-interpretation on G_4 can be used to generate all context-free grammars.

In the tuner, we want to be able to generate any context-free grammar. So it appears that any grammar form, in normal form, will serve our purpose. However, remember that any s-interpretation is length preserving. Therefore, we are not able to use any of the grammar forms presented thus far.

As we were able to find an equivalent attributed grammar from an attributed grammar with right regular parts, we can also find an equivalent grammar form which also uses right regular parts. With right regular parts, we would not have to worry about the length preserving

nature of s-interpretation because if we need a production rule whose length was longer than what we already had, we could increase our productions in the grammar form.

We call our ultimate grammar form genesis. Let this grammar form be $G = (v, \theta, P, \sigma)$ where P has the following format:

$$S ::= (a^* S^*)^*$$

It is easy to illustrate that any of the grammar forms that we have defined, can be defined by G . Let us take a couple of them and illustrate what we mean. All we have to do is show that they are strongly equivalent or that there is a grammar morphism $h: G_1 \rightarrow G$. Clearly, we have

$$T ::= TT \in S ::= (a^* S^*)^*$$

$$T ::= b \in S ::= (a^* S^*)^*$$

$$h(T) = \{S\}$$

$$h(b) = \{a\} \text{ which implies that } h(v_1) = v$$

G is both fine and length preserving: therefore the two grammars are strongly equivalent.

Let us take our more complicated example of grammar form G_4 . Again all that we need to do is show that the two grammars are strongly equivalent or that there is a grammar morphism $h: G_4 \rightarrow G$. Again, we clearly have

$$T ::= TTb^9 \in S ::= (a^* S^*)^*$$

$$T ::= TbTb^8 \in S ::= (a^* S^*)^*$$

$$T ::= bTbTb^7 \in S ::= (a^*S^*)^*$$

.

$$T ::= b^9TT \in S ::= (a^*S^*)^*$$

$$h(T) = \{S\}$$

$$h(b) = \{a\} \text{ which implies that } h(v_4) = v$$

G is both fine and length preserving; therefore the two grammars are strongly equivalent.

Theorem 12: Every grammar form can be generated by the "genesis" grammar form.

Proof: The "genesis" grammar rule represents a collection of grammar rules. Since we already know that we can make the vocabulary substitution, we will concern ourselves with only making sure that all grammar rules of two letters can be represented by the genesis grammar rule. Let our vocabulary be

- $v = \{a, S\}$,
- $\theta = \{a\}$, and
- $v - \theta = \{S\}$

We will prove the theorem by induction. We will first show that the genesis grammar will produce all grammars where the length of the right expression of all the production rules in that grammar are one. We will then assume that the genesis grammar will produce all grammars where the length of the right expression of all the production rules in that grammar are less than or equal to n . Then we will prove it for length $n + 1$.

For $k = 1$. For grammar rules with right expression of length one the grammar rules must look like:

$S ::= S$ or

$S ::= a$

Clearly both of these rules are in the set $(a^*S^*)^*$.

For $k = n$. Assume it is true.

For $k = n + 1$. For grammar rules with right expressions of length $n + 1$ the grammar rules will look like grammar rules of length n with one additional symbol.

$S ::= BS$ or

$S ::= Ba$ where

B is the right expression of a grammar rule of length n .

Since B is already known to be in the set $(a^*S^*)^*$, all we need to show is that BS and Ba are also in the same set. For simplicity, we will assume that for each symbol in B that we use exactly one symbol in a^*S^* . Therefore, B is in the set $(a^*S^*)^n$. BS is then in the set $(a^*S^*)^nS$ which is in the set $(a^*S^*)^{n+1}$ which is in the set $(a^*S^*)^*$. Likewise, Ba is in the set $(a^*S^*)^na$ which is in the set $(a^*S^*)^{n+1}$ which is in the set $(a^*S^*)^*$. End of proof.

We can see that our genesis grammar form can generate any of the grammar forms that in turn can generate all of the context-free grammars. This genesis grammar form will be used as our model for the generation of programming environments and also for the tuning of these environments.

4.7. Attributed Grammar Forms

The natural extension to our model should be the addition of attributes and semantic functions to grammar forms. This extension has been addressed by Soni in [96] and Kuo in [61]. However, their definition of an attributed grammar form requires that we have equivalence classes for both attributes and semantic function. These equivalence classes have not been studied and the inherent properties have not yet been addressed. Soni mentioned that we could have characteristic attributes and a constraining function that would do the mapping of attributes to semantic functions. However, for the present time we want to take a more restrictive notion of attributed grammar forms.

So our definition of an attributed grammar form is a grammar form where the context-free grammar is an attributed grammar.

In other words we are going to allow the set of attributes, their domain sets, and their evaluation rules to remain the same for each interpretation symbol and production. An issue for further research would be developing a model of attributed grammar form based on the definitions given by Soni.

With our simpler notion of attributed grammar form, we will formally define an attributed grammar form to be a quadruple $AGF = (G, V, \psi, \varsigma)$ where

- $G = (G_0, A_0, A, \text{sem})$ is an attributed grammar, often referred to as the form attributed grammar of AGF,
- V is an infinite vocabulary.

- $\psi \in V$ is an infinite vocabulary of terminal symbols,
- ζ is the allowable set of interpretations of G using the symbols in the two vocabulary sets.

With this definition, the set of attributes associated with the form grammar symbol pass directly to the interpreted symbol. The same is true with the set of semantic functions.

Chapter 5

TRIAD and the Grammar Form Model

The tuner is a TRIAD tool that takes a specific grammar form and creates a method representation to be used to customize TRIAD. The tuner itself is method driven with the meta method. The tuner works in one of two modes; static mode or dynamic mode.

In the static mode, the user builds a form tree using the meta method. This form tree is then sent to the method compiler, a part of the tuner, which creates the data structure representation needed by the new method. With this new method, the user can then customize the TRIAD system.

In dynamic mode, users use a partially defined method to customize the TRIAD system and instantiates a form tree from that method. Users then change mode from editing to tuning. In tuning mode, users are allowed to make changes to the structure of either the blank forms or the filled forms by manipulating the tags, the form representation, the attributes and the procedural components. When users have finished making changes deemed necessary, they signal TRIAD by leaving the tuning mode and moving back into editing mode. At this time TRIAD calls the method compiler again to verify the consistency of the method. If users like, at this point they can also specify that this is a new method and have it stored in the method database.

In this chapter we will discuss the relationship between the grammar form and the meta method and how the tuner uses this meta method to build methods for TRIAD. In section 5.1 we will look at how a method is described. Then in section 5.2 we will look at how the tuner is able to manipulate the tags, form representations, the attributes, and procedural components. Finally in section 5.3, we will see why the grammar form model is the best model for representing methods.

5.1. Description of a Method in TRIAD

As we have seen with other software environments, the description of the environment had two parts; the description of the grammar or attributed grammar and some type of display interface. Since TRIAD is method driven, it should come as no surprise that a method should be described by two parts; the description of the attributed grammar form and the description of a forms interface.

Formally, we then can define a software method (SM) as a triple (AGF, DSP, FI) where

- AGF = (G, V, ψ, ς) is an attributed grammar form,
- DSP is display representation of the symbols of G, and
- FI is a mapping of the grammar rules of G to a forms interface.

The best way to describe a method in TRIAD is to use an example. The example we will use is the meta method which is the method used by the tuner. The forms of the meta method are shown in figures 15, 16, 17, and 18. These forms are all blank forms. Project information is

META-FORM-1	Methodology	Form-use-#
Methodology Name:		
{2} Grammar Definition:	Form-use-#	
Form Definition [more?]:		
Form Number:		
Production Number:		

Figure 15: Methodology Form of the Meta Methodology

chunked according to the underlying concept grammar productions with associated attributes and procedural components. For example, underlying the Methodology Form (see figure 15), we have the following concept grammar production rules:

$$\begin{aligned}
 \langle \text{method} \rangle &::= \langle \text{name} \rangle \\
 &\quad \langle \text{grammar} \rangle \\
 &\quad \langle \text{form interface} \rangle^+ \\
 \langle \text{form interface} \rangle &::= \langle \text{form number} \rangle \\
 &\quad \langle \text{root node} \rangle
 \end{aligned}$$

Several concept grammar production rules are grouped together to constitute a "form" in TRIAD. A form presents a primary method concept to the user. Each production in the form then presents a subconcept of the method or a refinement of the primary concept. For example, using our Methodology Form again, the concept that is presented by the form is that a method has three parts - a unique name, a concept grammar description, and a form interface description.

META-FORM-2

Grammar Definition

Form-use-#

Grammar Name:

Action Set:

{3} Action [more?]: Form-use-#

Attributes:

{4} Attribute Set: Synthesized Local Form-use-#

{4} Attribute Set: Inherited Local Form-use-#

{4} Attribute Set: Global Form-use-#

Symbols:

Start Symbol:

{5} Symbol: Form-use-#

Heading:

Displayable Help Information:

Terminal Symbol:

{5} Symbol [more?]: Form-use-#

Non-terminal Symbol:

{5} Symbol [more?]: Form-use-#

Production Set:

{6} Production Rule [more?]: Form-use-#

Figure 16: Grammar Form of the Meta Methodology

META-FORM-6**Production Rule****Form-use-#**

Production Number:
Left Hand Symbol:
Right Hand Symbol [more?]: (given in left to right order)
Name:
Operation:
Kleene Star:
+:
0/1:
Form Representation:
Heading:
Displayable Help Information:
Prefilled Entry:
Entry Updatable:
Number of lines:
Semantic Functions:
Name [more?]:

Figure 18: Production Form of the Meta Methodology

This corresponds to the definition of a software method. For human engineering reasons, the display representation of a symbol is made a part of the grammar definition. Each of these parts of the method can be interpreted.

For example, in figure 19 we have an instantiated filled form for the meta method. Here the method name is "test.md". The concept grammar description is further refined in a grammar form which has a use number of 2. A subconcept in the method form would be - the form interface description. The form interface description says that there are two forms in this method. The first form begins with production 1 and the second form begins with production 3. The concepts do not change from one instantiation of the method to another, just the interpretations.

META-FORM-1	Methodology	Form-use-# [1]
<hr/>		
Methodology Name: test.md		
<hr/>		
{2} Grammar Definition:	Form-use-# [2]	
<hr/>		
Form Definition [more?]:		
<hr/>		
Form Number: 1		
<hr/>		
Production Number: 1		
<hr/>		
Form Definition [more?]:		
<hr/>		
Form Number: 2		
<hr/>		
Production Number: 3		
<hr/>		

Figure 19: An Instantiated Methodology Form

Now recall that we defined a grammar form F as a quadruple (G, V, ψ, ζ) where

- $G = (v, \theta, P, \sigma)$ is a context-free grammar,
- V is an infinite vocabulary,
- $\psi \in V$ is an infinite vocabulary of terminal symbols, and
- ζ is the allowable set of interpretations of G using the symbols in the two vocabulary sets.

The genesis grammar form has the following context-free grammar:

- $v = \{a, S\}$,
- $\theta = \{a\}$,
- $P = \{P_1 : \{S\} ::= \{a^* \{S^*\}^*\}$, and
- $\sigma = \{S\}$

That grammar is not shown in the meta method but is inherent in the tuner tool. The set of symbols shown in the Grammar Definition Form (see figure 16) is the infinite vocabulary V . This infinite vocabulary is defined by the start symbol, the set of terminal symbols, and the set of non-terminal symbols. The set of terminal symbols is also the infinite vocabulary of terminal symbol ψ . The set of productions shown in the Grammar Definition Form is the allowable set of interpretations of P_1 of G using the symbol set; the set of productions is ζ .

An attributed grammar was defined as a quadruple $G = (G_O, A_G, A, \text{sem})$ where

The second model is the attributed grammar model. Several programming environments had been built on this model as is shown in figure 8. One problem with this model was that all the project related information is only at the leaf nodes. Thus, the project related information string itself is not structured. This structure is maintained separately from the project information. When attempting to structure this information according to method concepts, the information must also be at the internal nodes. For example, suppose that in a method there was a concept of a meeting where:

$$\langle \text{meeting} \rangle ::= \langle \text{participants} \rangle \langle \text{action items} \rangle$$

In the attributed grammar form model, the minutes of the meeting could be associated with the 'meeting' node, the list of attendees with the 'participants' node, and the follow up items with the 'action items' node. In the attributed grammar model, another terminal symbol, 'minutes', would have to be added to the grammar to be associated with the minutes of the meeting. The relationship between minutes of the meeting and the follow up items would have a different connotation in the attributed grammar model than in the attributed grammar form model. In the attributed grammar form model, the follow up items comes from the minutes of the meetings, where in the attributed grammar model, the follow up items are independent of the minutes of the meeting. Another problem has to do with grammar transformation. It cannot be done in the attributed grammar model. There is no procedure for deciding what to do with the information that is associated at a leaf node when that node is changed to an internal node and a subtree is attached to the node. In the attributed grammar form model, the information associated at a leaf node does not change if that node is changed to an internal node or expanded.

5.3. Why Attributed Grammar Form Model

We have illustrated the power of the tuner which is based on the attributed grammar form model. A logical question that could be asked is "Could we have the same power if the tuner had used some other model?" The answer is no. In this section we will look at some of the other models and see why we could not have used them for the tuner.

The first model is the relational model. Several systems have been built on the relational model [65, 69, 79, 107]. One problem with the relational model was that context information had to be reconstructed when the project related information was stored as tuples. For example, Linton illustrated in his work that in order to reconstruct this information, the context information had to be known in the first place. The context information could be stored in the tuples which would make the retrieval of the information more efficient, but the size of the database would increase without giving an increase in power. Access to the database can only be done through cross products, joins, and projections. Another problem with the relational model was the propagation of synthesized and inherited attributed values. In the relational model numerous queries over the whole database have to be generated to accomplish the same propagation of attributed values that is inherent in the attributed grammar form model. The propagation of attributed values is not a part of the relational model. Another problem was the dynamic creation and modification of tuple definitions. Although the tuple definitions can be changed, they only can be done in a batch and off-line mode. A special program has to be written each time a tuple definition is changed to transform the database to reflect the new tuple definitions. The same program can not be used for any arbitrary change to the tuple definitions.

The global controller for the change history tool is a small routine as follows:

```
begin
    set mode = change history generator
    open change history file
    visit tree in depth first order
    call change history routine
    set mode = change history output
    open change history record file
    visit tree in depth first order
    close change history record file
    close change history file
end
```

At the symbol "change history" the following action routine is added:

```
if (mode == change history output) then
begin
    while (change history request number == request number)
    begin
        copy line of data to the buffer
    end
end
end
```

We add a new symbol called "change history" and add it to the documentation concept production rule as follows:

```

<documentation>::= <user manual>
                    <programmer's manual>
                    <functional specification>
                    <change history> *

```

Now suppose that the module production rule reads as follows:

```

<module>::= <input specifications> *
            <output specifications> *
            <variable specifications> *
            <line of code> *

```

To each of the symbols "input specifications", "output specifications", "variable specification", and "line of code", we add a new attribute called "change number". Also to each of these symbols we add the following procedural component:

```

if (mode == editing update) then
    symbol.change number = current change request number
if (mode == change history generator) then
begin
    place symbol.change number in change history file
    copy the data in the buffer to the change history file
end

```

META-FORM-2**Grammar Definition**

Form-use-#

Grammar Name:**Action Set:**

{3} Test.create: Form-use-#

{3} Test.visit: Form-use-#

{3} Test.enter: Form-use-#

{3} Test.update: Form-use-#

Attributes:

{4} Attribute Set: Synthesized Local Form-use-#

{4} Attribute Set: Inherited Local Form-use-#

{4} Attribute Set: Global Form-use-#

Symbols:•
•
•**Terminal Symbol:**

{5} Symbol [more?]: Form-use-#

Non-terminal Symbol:

{5} Symbol [more?]: Form-use-#

Production Set:

{6} Production Rule [more?]: Form-use-#

Figure 22: Locally Tuned Form

This change does not change the concept grammar production underlying the form, only its representation to the user.

If the tag of a refinement organizer is updated, the tag of the refinement form is also updated. As a result, we now have four new forms titled, Test.create, Test.visit, Test.enter, and Test.update. We could then use more tailored queries. Instead of a traditional query like...

Find an action form with action name = Test.create

We could use a more focused query like...

Find a form = Test.create

5.2.2. Tuning Example of Concept Rebinding at Method Use Time

Suppose we have just developed a new tool that will take a listing of lines of code with a change number and output a sorted list based on change numbers. For each change number, the output will show which modules were changed and what lines in those modules were changed. Management has decided that this tool is valuable and wants it to be integrated into the system. Let us also suppose that in the past that there had been no concern about a change history report, and so there was nothing in the current system to handle this request.

The current documentation section is set up with the following concept grammar production rule:

```

<documentation> ::= <user manual>
                    <programmer's manual>
                    <functional specification>
```

- the adding or deleting of the tag display representations and the forms interface with the concept grammar production rules.

The genesis grammar form allows the tuner to make these rebindings dynamically. All the changes to the bindings are local changes which, if they do have to be propagated over the interpreted tree, the tuner, with the aid of TRIAD, has access to. For example, let us suppose that we decided to change the number of lines on the screen for the tag "Displayable Help Information" in form Production Rule Form (see figure 18).

Since TRIAD stores both the grammar symbol and the production number with each node, there is no problem going through a form tree to find all the nodes with the grammar symbol 'displayable help information' and production rule number 'x'. Every time such a node is found in the form tree, that node display attributes are updated to the new value. If the change is to occur to only one node, the user can place the cursor at that node and tell the system to update the display attribute of just that node.

5.2.1. Tree Rebinding at Method Use Time Tuning Example

Suppose we want to make the Grammar Definition Form (see figure 16) more meaningful. Instead of just having "Action" show up as a tag on this form, we would like to have the actual "action name" show up as is shown in figure 22. We can do this by placing the cursor at the tag "Action" and using the tuning command of "change-node-heading". By restricting the application of this command to the current tag and doing it four times, we could get the tuned form shown in figure 22.

An even more powerful concept of evolution is supported by the tuner's ability to modify method descriptions. Tuning rebinds concepts to interpretation related attributes, procedural components, and tags. There are two types of method use time rebindings: tree rebinding and concept rebinding.

Tree rebinding is done when changes are made to the interpreted tree that do not affect the underlying concept grammar. These changes include:

- the changing of a tag for a chunk of project information in the interpreted tree,
- the adding or deleting of a procedural component for a tag in the interpreted tree,
- the adding or deleting of attributes for a tag in the interpreted tree, and
- the changing of the display representation and the forms interface for the interpreted tree.

Concept rebinding is done when changes are made to the underlying concept grammar. These changes include:

- the adding or deleting of a concept grammar symbol,
- the adding or deleting of concept grammar production rules,
- the adding or deleting of attributes and/or procedural components for the concept grammar, and

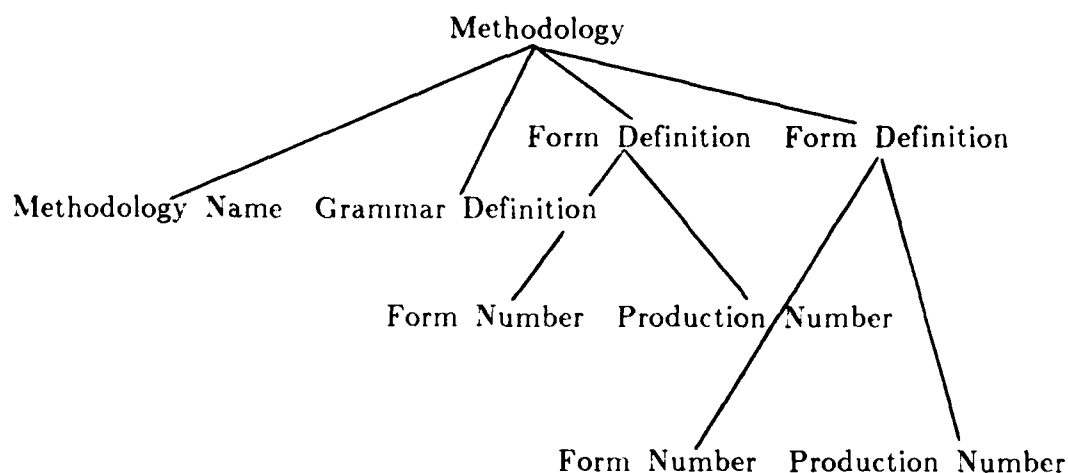


Figure 21: Concept Tree for an Instantiated Methodology Form

Notice that the two concepts trees are different. In essence, we can say that the concept grammar production rules have changed.

An interpreted concept tree is the filled form with all interpretable entries present. The Methodology Form in figure 19 is an example of an interpreted concept tree.

5.2. Tuning a Method

To contrast TRIAD with other systems (like ALOEGEN [74] or a parser generator [1]), it is useful to define two binding times; method define time, and method use time. A meta system like a parser generator is used to generate internal data structures which are interpreted by a generic parser. The language for the generated compiler is thus bound at grammar (language) define time. In contrast, the TRIAD meta system tools are bound to the description at method use time.

⟨Production Number⟩ }

The associations of procedural components and attributes always stay with the concept grammar symbol. The interpretation of them is always one for one with the interpretation of the concept grammar symbols and the concept production rules.

The next level of interpretation is the interpretation from the tag to the entry. This interpretation is not as clear.

The tuner makes the interpretation from the genesis grammar to the concept grammar and from the concept grammar to the blank form grammar. TRIAD helps the user make the interpretation from the blank form grammar to the project information database.

In order to keep definitions straight, let us tie the definition to what is shown to the user by the forms. A concept tree is a blank form when it is first instantiated. Notice that the concept tree is based on the blank form grammar and not directly on the concept grammar. The concept tree may change if an entry is repeated.

For example the blank Methodology Form in figure 15 has a concept tree which is shown in figure 20. The instantiated Methodology Form in figure 19 has a concept tree which is shown in figure 21.

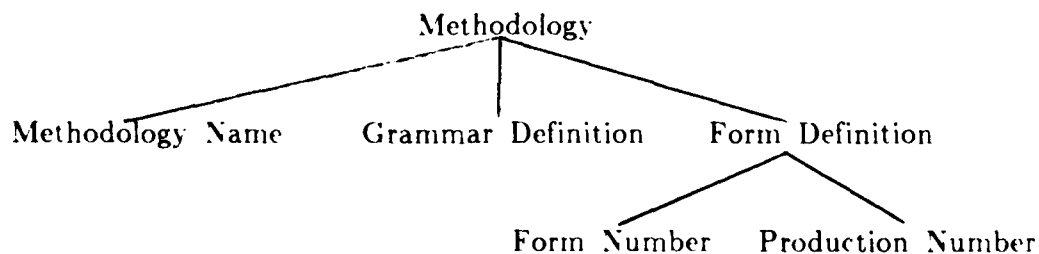


Figure 20: Concept Tree for a Blank Methodology Form

There are several levels of interpretation going on here. The first level of interpretation is from the genesis grammar to the concept grammar. There is also a level of interpretation going from the concept grammar to the grammar actually displayed in the blank forms. Let us illustrate this with the Methodology Form. The concept grammar production rule for the Methodology Form is:

$$\begin{aligned} \langle \text{method} \rangle &::= \langle \text{name} \rangle \\ &\quad \langle \text{grammar} \rangle \\ &\quad \langle \text{form interface} \rangle^+ \\ \langle \text{form interface} \rangle &::= \langle \text{form number} \rangle \\ &\quad \langle \text{root node} \rangle \end{aligned}$$

The following interpretations are used to come up with the blank form grammar:

$$\mu(\text{method}) = \{\text{Methodology}\}$$

$$\mu(\text{name}) = \{\text{Methodology Name}\}$$

$$\mu(\text{grammar}) = \{\text{Grammar Definition}\}$$

$$\mu(\text{form interface}) = \{\text{Form Definition}\}$$

$$\mu(\text{form number}) = \{\text{Form Number}\}$$

$$\mu(\text{root node}) = \{\text{Production Number}\}$$

$$\mu(\text{of the first production}) = \{\langle \text{Methodology} \rangle ::= \langle \text{Methodology Name} \rangle$$

$$\langle \text{Grammar Definition} \rangle$$

$$\langle \text{Form Definition} \rangle^+ \}$$

$$\mu(\text{of the second production}) = \{\langle \text{Form Definition} \rangle ::= \langle \text{Form Number}$$

figure 18) the values of "Number of lines" and "Entry Updatable" would be stored as attributes for the display interface of the form. The concept presented in this form is..

- actual production description.
- display interface.

How this form is presented on different terminals may vary, but concepts inherent in the presentation have at least two subconcepts: how much space does it take on the screen, and whether or not the user can change the interpretation. The values of "Number of lines" and "Entry Updatable" would be stored as attributes with the concept grammar symbols and each terminal type would access its appropriate attribute values.

Also associated with each production are procedural components. These procedural components could be fired by a global controller for a particular tool or fired based on the state of the system. For example, when the system is in editing mode, one set of procedural components could be fired when a node in the interpreted tree is visited or updated. Another set of procedural components could be fired when the system is in compile mode and the compiler interface (a global controller) is traversing the interpreted tree.

Procedural components can be used to enforce method constraints. For example, with the Jackson method, we could have a procedural component attached to the program body that would check that the input and output concepts have been developed before the program body concept is developed.

The collection of procedural components are the action sets shown in the Grammar Definition Form. Sem, the semantic function associator for the grammar, is done in both the Symbol Form (see figure 17) and the Production Rule Form (see figure 18). The reason that procedural components are associated with symbols is because we allow some analysis to be performed on the interpretation of nodes. Non-terminal nodes can have interpretations just like terminal nodes. The associator was placed in the Symbol Form and the Production Rule Form for human engineering reasons only.

The display representation of the symbols is defined in the Production Rule Form (see figure 18). Each symbol has a 'heading' or 'tag'. This really can be thought of as an interpretation of the grammar symbol. There might be an interpretation already assigned to this tag in which case the interpretation is a prefilled entry. The 'displayable help information' provides for having help information appear in the panel of the form to aid users with filling the forms. If a tag is not to be interpreted or the preassigned interpretation is not to be changed, then the entry is flagged not to be updated. The 'number of lines' is the amount of screen space allowed for this symbol in the form. It does not mean that all the data has to fit into this amount of space, the entry can be scrolled.

We have allowed for right regular parts in our context-free grammar. For this reason the Operation Set has to be determined in the Production Rule Form. Only one of the operations may be assigned to the symbol in that production.

Although we have not shown attributes, they do play an important role in a method description. For example, in the Production Form (see

- $G_O = (v, \theta, P, \sigma)$ is a context-free grammar,
- A_G is a specification of attributes,
- A is an attribute associator for G and A_G , and
- sem is a semantic function associator for productions in G_O such that $\text{sem}(i)$ is a valid collection of semantic functions for production P_i in P .

The context-free grammar we have already identified. The set of attributes shown in the Grammar Definition Form is A_G the specification of attributes. For human engineering reasons, we allow global attributes.

Global attributes are attributes assigned to the root node of the tree that all nodes throughout the tree want to have access to. Instead of making this collection of attributes a part of each node and using the copying function to move the values around the tree, they are assigned to a specific spot that all the nodes know about. Also, since we are using time-varying attributes, they must be allowed to have initial values. The definition of the attribute sets are shown in the Attribute Set Form (see figure 17). The concept 'type' shown in the Attribute Set Form is an implementation detail and is there so that the computer can interpret the default value.

The attribute associator A is done in the Symbol Form (see figure 17). It was placed in this form for human engineering reasons. Recall that the attribute associator assigns attributes to symbol. Again we are using time-varying attributes and so the default value of the attribute must be specified.

The last model is the frame model [86]. This model is used in artificial intelligence systems. Although project related information can be stored in the internal nodes (called slots), there are no grammar symbols associated with the slots. All that can be stored in a slot is project related information and attributed values. It is difficult to build generic queries without some type of generic tags or symbols on the slots. In this model there are only inherited attributes; there are no synthesized attributes. The propagation of attributed values down the tree can only be stopped explicitly in the frame model. Attributes for method concepts need to be more flexible than what the frame model allows. Attributed values in the frame model can not flow across the tree like they can in the attributed grammar form model and which is need for representing method concepts. Like with the other two models, dynamic transformation of instantiations of the model are impossible. Since the slots only have a value and not a tag, a generalized program to manipulate the slots can not be written.

Based on these four models, the best model to model the concepts of a method is the attributed grammar form model. This is because

- The attributed grammar form can have grammar transformation which is need in order to reflect experience gained with the use of the method.
- The attributed grammar form can have information stored in the internal nodes which allows the modeling of stepwise refinement.
- The attributed grammar form can attributes that are not forced to pass information completely up and down the tree.

Chapter 6

The Tuner

One of the best techniques to test out the meta method is to use the meta method and bootstrap the representation of the meta method. We therefore decided that we would use the tuner to generate the meta method. We did this by manually generating the grammar symbols and productions and a forms interface that the static tuner would understand. We could not use the previous version of TRIAD as that version did not have a data structure for procedural components and attributes. With this manually generated version of the meta method, we brought up the first tuner. Then with this tuner we brought up the static tuner with its capability to define attributes and procedural components. With this tuner we could now do the consistency checks that will be explained later. With this new tuner, we brought up the next version of the meta method. This version of the meta method was still without attributes and procedural components. The dynamic tuner was then developed. With the dynamic tuner, we then brought up the final version of the meta method which included attributes and procedural components. The details of the grammar production, attributes, blankforms, and procedural components can be found in Appendix A.

In section 6.1, we will discuss the meta method. We have already discussed the model on which the meta method is built, but we still need to discuss the steps involved in the method. Then in section 6.2, we will discuss some of the implementation issues involved in bringing up the meta method using the dynamic tuner.

Three terms that are closely related need to be defined. Semantic function was formally defined in section 4.4. Procedural components are similar to semantic function except they can be used to update more than the local attributes. A procedural component can update an attribute in the derivation tree as well as provide an interface for tools. Action set is a collection of procedural components and the name used in the actual TRIAD implementation. These three terms will be used interchangeably where the distinction between the terms is not important.

6.1. Description of the Meta Method

The meta method is a procedural way of defining the attributed grammar form and display interface for TRIAD. Although there are steps in this method, these steps do not need to be followed in exact order because of the incremental design of the tuner. We will first present the steps of the method and then present a brief discussion on how to build a method using the meta method.

The meta method is broken up into two parts: a definition part and a compilation part. The definition part has five steps and the compilation part has one step.

1. Define Symbol Set. In defining the symbol set, we are really defining the vocabulary v and start symbol σ of the grammar form. Recall that the vocabulary is broken up into two sets, namely the terminal vocabulary and the non-terminal vocabulary. The non-terminal vocabulary could be an empty set. From here on in, we will use the terms vocabulary and symbol interchangeably. In defining the symbol, we give it a name and use the attribute associator to give it its set of

attributes and the semantic function associator to give it its set of procedural components. If a symbol has a set of attributes and/or procedural components, they must be defined before they are associated with the symbol.

2. Define Production Rule Set. In defining the production rule set, we really are defining the μ functions of the genesis grammar form to transform to the new grammar form. Each symbol in a production has a flag indicating its repetition status and a display interface. The repetition status is our use of the right regular grammars. In defining the display interface, we must specify what the symbol will be known as in the form, what help information is needed for interpreting the node represented by the symbol, what prefilled value the interpretation of the symbol may have, a flag to indicate if we can change the prefilled interpretation, and the space on the actual screen this symbol with its interpretation will be allowed to have. Each symbol in the production must be defined before it is referenced, and each left hand symbol of the production must be from the non-terminal set of symbols.
3. Define Attribute Set. The attribute set for the genesis grammar form is mapped one-to-one to the attribute set of the generated grammar form. There are three classes of attributes that need to be defined, namely, global attributes, synthesized attributes, and inherited attributes. Each attribute has a name, a type, and a default value. The type is the data structure associated with the attribute. Default values for global attributes must be defined at this time. The

default values of the other two classes of attributes can be delayed until they are associated with a grammar symbol.

4. Define Action Set. The action set for the genesis grammar form is mapped one-to-one to the action set of the generated grammar form. Therefore, in order to define this set, the user simply defines unique action names and the associated action procedure.
5. Define Blankform Set. This step is involved in defining the forms interface. A blankform has a form number and a production number of the root production of the form. The boundaries of the form do not need to be defined. The boundaries of the forms are calculated by the system when all of the forms have been defined.
6. Compile Method Description. This is an automatic step done when the user indicates that the description of the method is complete. In this step, the method description is checked for consistency, like "have all the productions been assigned to a forms interface (although they do not need to be assigned as a root production of a form)" and "are there any recursive productions in a form".

6.2. Implementation Issues of the Meta Method

The meta method was the first example to be brought up. In section 6.2.1 we will discuss the issues associated with procedural components in general as well as those with the meta method. In section 6.2.2 we will discuss the issues associated with attributes in general as well as those

with the meta method. The issues associated with symbols, production rules, and display interface are not unique to TRIAD and have been discussed with such systems as CPS and ALOE. For that reason we will not rediscuss them in this dissertation.

6.2.1. Implementation Issues of Procedural Components

Each procedural component has three parts; a name, a procedure, and a set of criteria for when the procedural component is executed. The user specifies an unique name. Normally, a procedure should have a set of arguments associated with it. However, we found that the only argument that we could allow was the current node. This was because there was no way for the procedural component firing routine to know which procedural components have arguments and where their arguments are stored. If a procedural component needs any arguments, they have to be stored as attributes accessible from the current node. The implementation and referencing of attributes will be discussed in section 6.2.2.

A language for defining the procedure had to be selected. Since we were only building a prototype, it was decided to use an interpretative language which could be brought up quickly. A lisp like language was developed and several elementary functions were defined so that the user could get to the node structure and the attributes. Two of the more advance functions were the GetNodeAtb and SetNodeAtb function which allows the user to retrieve and store attributes for any node in the tree. Given the node-id and the attribute name at the requested node, these routines would access the address of the attribute and perform the appropriate operation. The disadvantage of using a lisp like language is that whenever the procedural component is fired, the procedure has to be

interpreted again. An area for future research is the development of a compiler for our lisp like language which can delay the binding of the attributes until execution time.

The next issue had to do the set of criteria for when the procedural component is executed. This is called the firing mechanism. In ALOEGEN, the action routines were fired each time the node was accessed. Then, within the action routine, a decision was made on why the node was accessed and an appropriate portion of the action routine was executed. We decided that it might be more appropriate to make the firing mechanism associated with the attribute name. In this way, we would not have to interpret a procedural component if it did not have to be fired. The firing mechanism then became an extension to the procedural component name, which a local controller checks to see if the procedural component needs to be fired. We selected six condition under which a procedural component could be fired.

1. Create a Node. A procedural component could be fired when a non-terminal symbol was expanded by a production rule. Under this condition certain attributes could receive initial values based on the attribute of the parent node.
2. Delete a Node. A procedural component could be fired when a node is about to be deleted from the form tree. Under this condition certain attributes of the parent node could be updated to reflect the deletion of this child.
3. Repeat a Node. A procedural component could be fired when a node is repeated in the form tree. What happens is the grammar symbol of a node is repeated and a new node is

created. Under this condition both the parent node attributes could be updated as well as the sibling node attributes being allowed to set up links the his sibling.

4. Enter a Node. A procedural component could be fired each time the node is entered. Upon entry, the node could access attributes of other nodes and set up the environment in which the node could be interpreted.
5. Update a Node. A procedural component could be fired when the user leaves the node and the entry has been modified causing a new interpretation to be generated. Attributes which store status information could be updated.
6. Leave a Node. A procedural component could be fired when the user leaves the node and the entry has not been modified during this visit. If the node interpretation is not modifiable, then firing upon exit would have the same affect as firing upon entry.

Another issue is what to do when a procedural component detects an error. For example, suppose that an entry is to contain an integer but the procedural component has determined that it doesn't. The procedural component needs to signal the TRIAD system what it wants to do with this error. It could let the error stand as is or it could require that the error be corrected before it continues on. To accommodate this signaling requirement two exiting functions were developed. The function Return allows the system to continue regardless of the error while the function Quit requires the user to correct the error before continuing on.

Another issue is how to display summary information to the user in the form. The easiest way to handle this issue is to have functions in the lisp like language that allow the user access to the interpretation of certain nodes. This feature became very handy when we wanted to place information in the parent form that is in the child form. For example, when a symbol name was defined in the Symbol Form we want that same name to appear in the Grammar form in the entry where the Symbol Form was refined. This is illustrated in figure 23.

Procedural components for the meta method take on the form of a consistency checker. There are two types of consistency checkers: one verifies that a name does not appear in a table and the other verifies that a name does appear in a table. An example of the first type of consistency checker is when a new symbol name is input, the procedural component, *symbol-name.update*, checks the list of symbols to verify that *no other symbol has that name*. This algorithm is shown in figure 24. An example of the second type of consistency checker is when a semantic function is associated with a production, the procedural component, *saction.update*, checks the list of actions to verify that an action with this name has been defined. This algorithm is shown in figure 25.

The actual details of the procedural component for the meta method are given in the listing found in Appendix A.

6.2.2. Implementation Issues of Attributes

Each attribute has a name, a class, a type, and a value. The name must be unique but it could be the same name as a symbol. We allow

META-FORM-2	Grammar Definition	Form-use-#[2]
Grammar Name:book.grm		
	•	
	•	
	•	
Terminal Symbol:		
{5}	Symbol [more?]:title	Form-use-#[4]
{5}	Symbol [more?]:author	Form-use-#[5]
{5}	Symbol [more?]:introduction	Form-use-#[6]
{5}	Symbol [more?]:main points	Form-use-#[7]
{5}	Symbol [more?]:conclusions	Form-use-#[8]
Non-terminal Symbol:		
{5}	Symbol [more?]:chapter	Form-use-#[9]
{5}	Symbol [more?]:chapter form	Form-use-#[10]
{5}	Symbol [more?]:section	Form-use-#[11]
{5}	Symbol [more?]:section form	Form-use-#[12]
Production Set:		
{6}	Production Rule [more?]:1	Form-use-#[13]
{6}	Production Rule [more?]:2	Form-use-#[14]
{6}	Production Rule [more?]:3	Form-use-#[15]
{6}	Production Rule [more?]:4	Form-use-#[16]

Figure 23: An Updated Fillform of the Grammar Form


```

Get the value of the entry
If (value is not proper type)
begin
    Tell user what the problem is
    Allow the user to make correction
end
Find the corresponding table
If (the table is empty)
begin
    Add this value to the table
    Return
end
For (each entry in the table)
begin
    If (this node did not create this table entry)
    begin
        If (value == entry.name)
        begin
            Tell user what the problem is
            Allow the user to make correction
        end
    end
    else
        Change the entry.name to value
end
end

```

Figure 24: Verify Name Not in Table Algorithm

this because the data structure for the attribute is different than for a symbol and the user has already identified which is which. We have already discussed the three classes earlier. We currently allow three types of attributes, namely, integer, string, and a list or table of integers. This table of integers was created so that we could access a collection of something. Our first use was to store node-ids of where symbols, actions, productions, forms, and attributes were defined. The types of attributes allowed must be consistent with the types of data structure allowed in our lisp like language. If we add a new attribute type, we also need to add that data structure type to our lisp like language.

```

Get the value of the entry
If (value is not proper type)
begin
    Tell user what the problem is
    Allow the user to make correction
end
Find the corresponding table
If (the table is empty)
begin
    Clear the entry
    Tell the user what the problem is
    Return
end
For (each entry in the table)
begin
    If (entry.name == value)
        Set a flag that says the name was found
end
If (the name was not found)
begin
    Clear the entry
    Tell user what the problem is
    Return
end

```

Figure 25: Verify Name is in Table Algorithm

Our first issue was how to access the attribute in our procedural components. We wanted to be able to get to the attributes of the current node, the parent of the current node, the children of the current node and the global attributes. We decided that one way to access these attributes was to have the attribute name as an extension to a symbol name. For example, if we want to get to the attribute, whose name is time, of the parent node, whose symbol name is schedule, we would reference it as schedule.time. Then it was the responsibility of the local controller to verify that indeed the parent of the current node was schedule and it did have an attribute named time. Global attributes

would be reference only by the attribute name. Doing this required that our lisp like language then had to have all of its temporary variables declared. Any attribute that was not a global attribute or an attribute associated with the current node, the parent node, or the children node, had to be accessed through the two functions `GetNodeAtb` and `SetNodeAtb`.

Another issue was which node was going to have the global attributes. They had to be tied to a node so that the interface would be simple, but which node do you tie it to? If it was tied to the root node of the form tree, it would be difficult to differentiate between the global attributes and the attribute that were really associated with that node. We decided that we would create a global node not a part of the form tree but associate with the form tree. All the global attributes would then be associated with this global node. This node would not have an id or any procedural component or a symbol associated with it.

Another issue was how to go about assigning default values to attributes. An attribute is a place holder for a node. In other words it had to be allocated space with each node. This is unlike the symbol name associated with each node. A node could simply have a pointer to the symbol because the node never updates the symbol data structure. When defining an attribute, we could give it a default value which would be its initial value. When the attribute was associated with a symbol, that default value could be changed by the user. Therefore, we had to copy the attribute when we associated it with a symbol, instead of just saving a pointer to it. When the attribute became a part of the node, we had to copy it again because each node needs to change the value of the attribute as the procedural components were executed.

Chapter 7

Using the Tuner for a VLSI Method

The VLSI domain has developed several well defined, highly evolved methods [106, 13, 100, 95, 39]. Many of these methods have been developed for a specific technology. Some have even been built around the tools that were available. The VLSI method that we developed was based on experience gained from the use of a set of CAD tools from the University of California at Berkeley. These tools included a graphics editor [78], a circuit extractor, programmable logic array generation tools, switch level simulators, plotters, design rule checkers, and analyzers running under VAX-UNIX. The VLSI method is illustrated in figure 26.

In section 7.1 we will briefly describe the steps of the method. Then in section 7.2 we will discuss how the tuner was used to dynamically build the representation of this method.

7.1. Description of the VLSI Method

The VLSI method provides a way for recording the inputs to various CAD tools and propagating the files from one CAD tool to another. This method was designed for student use to get them acquainted with the CAD tools. Therefore only a portion of the total CAD tools was included in the method design. The method consisted of describing some state diagrams and using those diagrams to drive the CAD tools until a logic level simulation was completed.

- determine the functions provided by that member,
- and define the conceptual objects and operations for that member...,

and then work with the next another member of the domain. Therefore, steps 1, 2, and 3 were done in order once for each member of the domain. The blankforms associated with this are shown in figure 30.

Another problem had to do with the attribute data types that were available. TRIAD currently only allows three types of attributes, namely: integer, string, and a list of integers. Although these types are powerful to express most attributes, it became difficult to compute the intersection set of all the object/operation sets. It was impossible to calculate the union set of all the object/operation sets because a list of strings was needed to store this set. Without the union set, we could not calculate the difference set.

A powerful idea that was used in this implementation was the ability to access attributes that were not tied to nodes that were immediate ancestors or descendants of the node or global attributes. This was best illustrated with the calculation of the intersection set. The attribute set for the domain consisted of an attribute of type "table of integers" where the integers represented the node-ids of each of the domain members. The attribute set for the domain members consisted of an attribute of type string containing the domain member name and an attribute of type table of integers where the integers represented the node-ids of each of the object/operation sets that were included in that domain. The attribute set for the object/operation set consisted of an attribute of type string containing the object name and an attribute of type table of

15. Perform Test. Using the test designed in step 8.1, perform the verification test for the virtual interface. This step can be automated.
16. Prepare Documentation. Prepare the necessary documentation. If each step has been properly documented, this step can be automated.

8.2. Implementation Issues of the Virtual Interface Method

The implementation details of the virtual interface method are found in Appendix C. It includes such details as the concept production rules, the blankforms, and some of the procedural components. The discussion in this part of the dissertation concerns conceptual details of the implementation.

The virtual interface method was brought up in 100 person hours. The static tuner was used to create the first production and the first blankform. The rest of the method was brought up using the dynamic tuner. This allowed the method designer to visually see the changes that she was making.

The implementation of the virtual interface method was not as straight forward as the method might indicate. One of the first problems was how to present the forms to the user so that meaningful chunks could be viewed at a time. With the size of the terminal screen available, it was decided that smaller chunks of data were better to present to the user. So instead of doing steps 1, 2, and 3 in that order, it was decided to...

- identify a member of the domain.

9. Specify the Interface. Each function which is to be included in the virtual interface must be specified by the selected specification technique. This step can be automated.
10. Design Test. Design a test to verify the virtual interface. This is a test of the interface and not a test of whether the individual packages work. This step can not be automated.
11. Split Interface. Split the interface into two parts - target independent and target dependent. The target independent part includes functions of a domain member which will be included in the interface. The target dependent part includes functions (which have to be supplied) to be included in the interface. This step can be automated.
12. Choose Implementation Languages. Choice of the language or languages for the implementation must be considered in terms of the programming environment and the goals of the project. This step can not be automated, but if the programming environment has a transformation system some of the following steps could be automated.
13. Implement Target Independent Part. Implement the target independent part of the interface so that it can be easily extended. This step can be automated.
14. Generate Target Dependent Part. Generate the modules needed to implement the target dependent parts. Although this step can not be totally automated, it is possible to assist the user in developing the code for these modules.

which objects and operations are available for all members of the domain. This step can be automated and is basically an intersection of all the objects and operations of the members of the domain.

5. Determine Extent of Mismatch. Using the definition of the conceptual objects and operations, determine which objects and operations are not available in all members of the domain. This step can be automated and is basically the difference between the union of all the objects and operations of the members of the domain and the intersection of all the objects and operations.
6. Evaluate Effort to Implement. Evaluate the effort required to implement the virtual interface. The effort to implement the common objects and operations is easy to calculate. The effort to implement the non-common objects and operation is not so easy. This step can be automated for the common objects and operations.
7. Select Objects and Operations to Include. Based on the amount of effort that can be expended, determine the objects and operations to be included in in the virtual interface. Although this step is rather easy to do, it can not be automated.
8. Select Specification Technique. Select a specification technique or method for describing the objects and operations in a formal manner. This step can not be automated, but if the specification technique is rigorous, the following steps could be automated easily.

The method itself is a two part procedure. The first part consists of six steps which analyze the domain and provides an evaluation of the magnitude of the project. This is basically an informal approach since most packages are not based on formal definitions of the abstract objects. The second part consists of ten steps which define a more exact specification and implementation of the virtual interface. We will briefly look at each step and at how they could be automated.

1. Identify Domain. Identify all the application packages that should be included in the domain. This could be a difficult task. Making the domain too broad could make the implementation of the interface too costly. Making the domain too specific could make the implementation totally redundant. This step can not be automated.
2. Determine Functions Provided. Determine all the functions provided by the individual members of the domain. Care must be taken to insure that the functions are only identified, not how they are implemented. This step can not be automated.
3. Define Conceptual Objects and Operations Informally. Define all the conceptual objects and the operations on those objects informally, indicating which domain members contain them. This task could be difficult if the domain does not have an agreed upon set of objects and operations. This step can not be automated.
4. Identify Common Objects and Operations. Using the definition of the conceptual objects and operations, determine

Chapter 8

Using the Tuner for a Virtual Interface Method

Many methods have been developed over the years. One area where methods seem to be more clearly specified is in the defining of interfaces to existing systems. Some of these methods include a method for the Box [16], a method for Cousin-Spice [44], and a virtual interface method [24, 25]. We decided a good test case would be to implement an interface method. We selected the virtual interface method to use the tuner on because the method itself was well described and we were already familiar with it.

In section 8.1 we will briefly describe the steps of the method. Then in section 8.2 we will discuss how the tuner was used to dynamically build the representation of this method.

8.1. Description of the Virtual Interface Method

Many computer systems have packages that perform similar functions. A virtual interface presents to the user a uniform view of a function which is independent of individual packages. These virtual interfaces attempt to include all the necessary and desirable functions of the equivalent packages and to present the best possible view to users of the interface within the constraints of the environment. The virtual interface method is a scientific approach in describing what is to be included in the interface and how that interface is to be built.

```
If (no flags have been changed since last time)
    Return
If (the node is not being refined)
begin
    Clear output file type flags
    Return
end
Translate the flags to appropriate command option for execution
Store that option in the shell command string
Determine which refinement form is being selected
Store the appropriate command option in the shell command string
Execute the shell command string
```

Figure 29: Algorithm for Tool Interface

$\langle \text{mextra1} \rangle$

$\langle \text{mextra2} \rangle$

$\langle \text{mextra3} \rangle$

$\langle \text{mextra4} \rangle$

$\langle \text{esim1} \rangle$

16. $\langle \text{type} \rangle ::= \langle \text{mextra} \rangle$

19. $\langle \text{cif} \rangle ::= \langle \text{mextra} \rangle$

Since we could not know for sure what the symbol name of the parent node, we needed to again add a new function to our lisp like language to get to the attributes of the parent node. A pointer to the parent node is maintained in the node structure. Once we find the node-id of the parent node then we can use the `GetNodeAtb` and `SetNodeAtb` to access and store the attribute values of the parent.

Most of the procedural components set flags that were passed to the node that was doing the actual interface to the CAD tools. The nodes that interfaced with the CAD tools had the algorithm shown in figure 29 implemented as a procedural component.

VLSI-FORM-3

EQNTOTT

Form-use-#[]

Input Filename:

Output Filename:

Output Truth Table:

Allow Input Names To Be Same As Output:

Output Variables May be Used in Expressions:

Reduce Truth Table:

No Redundant Minterms:

Human Readable Truth Table:

Output Number of Inputs:

Output Number of Product Terms:

Output Number of Outputs:

{4,5} Options:

4. Generate truth tables for TPLA

5. Compile EQNTOTT and pipe through TPLA with standard options

Form-use-#[]

Figure 28: EQNTOTT Blankform of the VLSI Method

the parent node. For example, there are three possible symbol names of the root node of the MEXTRA Form as is shown in the subset of the production rules of the VLSI method:

12. $\langle \text{output} \rangle ::= \langle \text{mextra} \langle$ 13. $\langle \text{mextra} \rangle ::= \langle \text{mextra-input} \langle$ $\langle \text{mextra-output} \rangle$

node. To overcome this problem we had to introduce pseudo-production rules to simply define new attribute values. In production rule part of Appendix B, there are several examples of these pseudo-production rules.

VLSI-FORM-2

PEG

Form-use-#[]

Program:

Output Filename:

Options:

Truth table filename:

{3,4,5} Output:

3. Generate equations for EQNTOTT
4. Compile PEG and pipe through EQNTOTT with standard options
5. Compile PEG and pipe through EQNTOTT and TPLA with standard options

Form-use-#[]

Figure 27: PEG Blankform of the VLSI Method

The next implementation problem had to do with finding out what choice the user had made on these nodes that had alternatives as refinement forms. For example, in the PEG Form with the "Output" node, how could a procedural component know which of 3, 4, or 5 the user had selected. The information is not stored in the interpretation. TRIAD stores the production number of refining productions in the node that is being refined. New functions then had to be added to the lisp like language to retrieve this production number from the node and make it available to the user. The use of this function is shown in procedural component output.visit in Appendix B.

Another implementation problem had to do with the symbol name of

7.2. Implementation Issues of the VLSI Method

The implementation details of the VLSI method are found in Appendix B. It includes such details as the concept production rules, the blankforms, and some of the procedural components. The discussion in this part of the dissertation concerns conceptual details of the implementation.

The VLSI method was brought up in 30 person hours. The static tuner was used to create the first production and the first blankform. The rest of the method was brought up using the dynamic tuner. This allowed the method designer to visually see the changes he was making.

One of the first problems encountered had to do with the form interface. In the meta method all nodes that could be refined had only one form to refine them. Therefore the display attributes of that node could also be the display attributes of the first node of the refining form. For example, in the Grammar Definition Form (see figure 16) the three types of symbols that are refined by the Symbol Form (see figure 17) have the tag "Symbol". These nodes cannot be reinterpreted and have a display height attribute of two. In the Symbol Form the tag is "Symbol", it cannot be reinterpreted, and its display height is two. This was not the case with the VLSI method.

For example, in the PEG Form (see figure 27), the node "Output" can be refined by one of three forms. Even though the node cannot be reinterpreted, it does have a display height of eight. One of the forms that can be used to refine it is EQNTOTT form (see figure 28). Its tag is different and its display height is only two. Therefore, we could not have the node's display attribute passed directly to the refining form root

designed circuits. After the design rule check is completed, send the final circuit to MEXTRA to create the circuit description.

5. Create a Circuit Description. Using MEXTRA read the circuit layout description and create the circuit description. From this circuit description, various electrical checks can be performed. MEXTRA allows the user to specify circuit scaling and capacitance capabilities.
6. Perform Electrical Checks. Using ESIM perform logic level simulation. ESIM is an event-driven switch level simulator for nMOS transistor circuits. From the result of this simulation, specify the changes that need to be made to the state diagram and go back and make the change.

This method does not force the user go from step 1 to step 6 in order. If the user has already defined some files that are appropriate for other than step 1, he could begin at the step in which his files are appropriate input files. Also, if users simply want to use the default options, they could at the appropriate step specify that they are using the default option for that next step and the method will take them automatically through that step to the next after that. Four additional steps could have been added to the method to take care of the SPICE program, the ERC program, the POWEST program, and the CRYSTAL program. The students were not expected to use those tools and so they were not included in this version of the VLSI method.

We will briefly look at each step of the method and the tool associated with that step.

1. Describe State Diagram. Describe the machine using the Moore model for finite state machines. This description consists of a list of the input signals, a list of the output signals, and a list of the state definitions. Then specify the options for the PEG program. The PEG program will generate a set of equations in the "eqn" format for the EQNTOTT.
2. Generate Truth Table Equations. EQNTOTT generates a truth table suitable for PLA programming from a set of Boolean equations which define the PLA outputs in terms of its inputs. In this step the user specifies the desired outputs and the restrictions on the truth table entries. This is done by specifying the options for the EQNTOTT program.
3. Generate Programmable Logic Array. From the output of EQNTOTT, TPLA generates PLAs in several different styles and technologies. The user is allowed to specify the style and technology and the restrictions for the AND and OR planes. The user also specifies whether the generate output can go to CAESAR for merging with other circuits or to MEXTRA to create the circuit description.
4. Merge and Optimize PLAs. Specify the PLAs that are to be included in the final circuit and connect those PLAs using the CAESAR editor. Then run LYRA on the final circuit. LYRA performs hierarchical layout rule checks on CAESAR

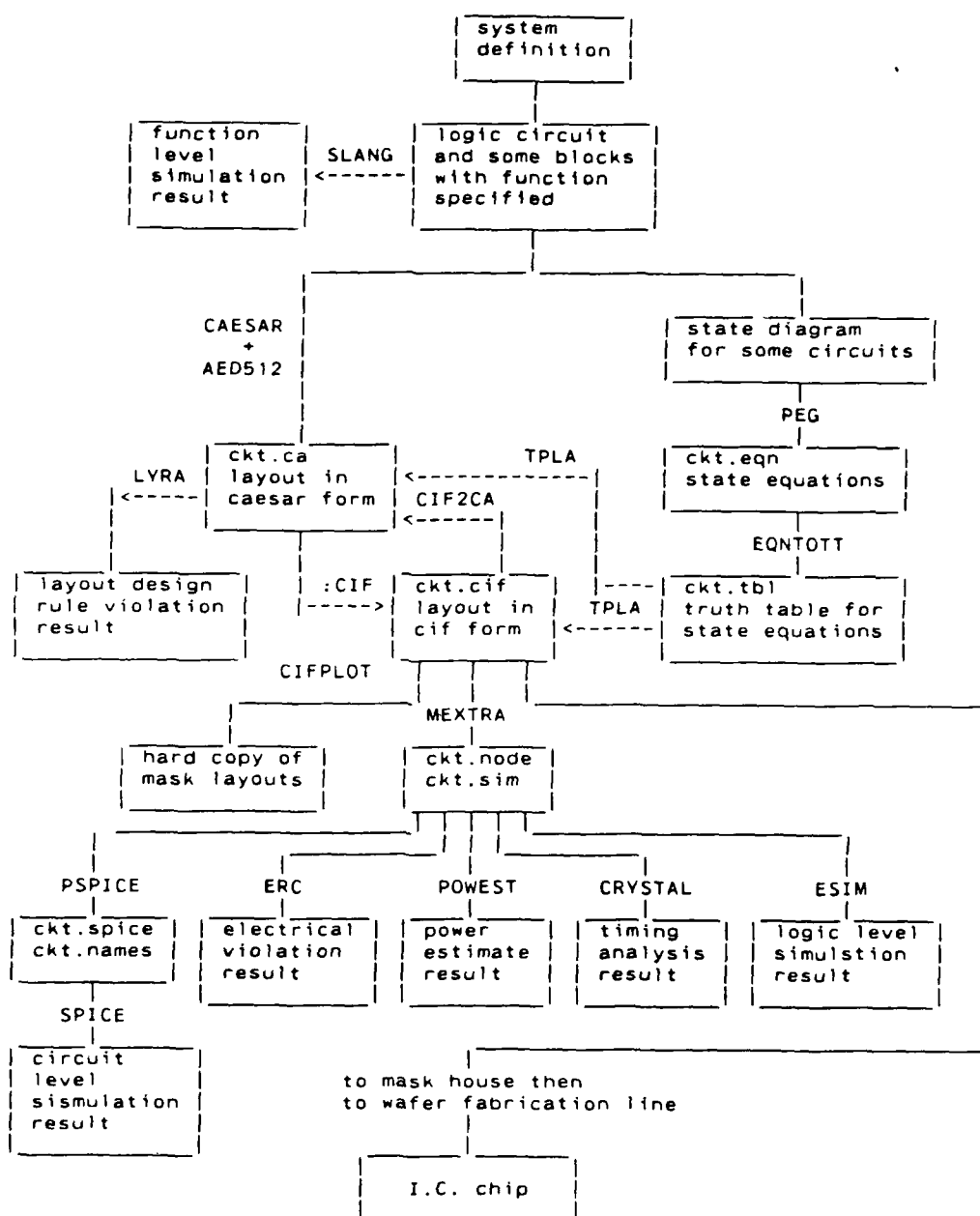


Figure 26: Diagram of VLSI Method

VIMETH-FORM-2 **Identify domain** Form-use-# []

Name of Domain: _____

{3} Domain member [more?]: Form-use-# []

VIMETH-FORM-3 **Functions Available** Form-use-# []

Member name: _____

{22} Function Set: Form-use-# []

{4} Object/Operation Set [more?]: Form-use-# []

VIMETH-FORM-22 **Function Set** Form-use-# []

Member Name: _____

Function [more?]: _____

VIMETH-FORM-4 **Define Object and Operations** Form-use-# []

Member Name: _____

Object name: _____

Operation name [more?]: _____

Figure 30: Define Domain Forms for Virtual Interface

integers where the integers represented the node-ids of each of the operations associated with this object. The attribute set for the operation is an attribute of type string containing the operation name. To calculate the intersection set of all the object/operations sets, an attribute of type integer was added to both the attribute sets of object/operation and operation. This attribute kept the count of the number of other sets that either had the same operation name or object name. By using the table of integers and the tree traversal routine of TRIAD, we can get or set any attribute value.

The procedural components for the virtual interface method were more of the analysis type, as compared with the tool interface type of the VLSI method or the consistency checking of the meta method. Typical of these analysis type procedural components is the calculation of the intersection set shown in figure 31. The actual code is found in Appendix C in the routine named identify-common.enter.

Two print procedural components were also written. The first was to print the list of all object/operations that were common. Because of the enforcement policy that was designed into this method, it was impossible to print the common list until after the intersection set had been calculated. To print this list, the algorithm simply selected one of the domain members and printed out all the object/operation names where their counts were equal to the number of domain members. An example of this is shown in figure 32. The actual code is found in Appendix C in the routine named common.enter. The other print procedural component written was the list of object/operations that were not common in all the object/operation sets. To print this list, the algorithm went through each list and checked the counts of the object

```

For (A = each object/operation set)
begin
  For (B = each object/operation set after A)
  begin
    If (object name of A == object name of B)
    begin
      Increment counts of both object names of A and B
      For (C = each operation in A)
      begin
        For (D = each operation in B)
        begin
          If (operation name of C == operation name of D)
            Increment counts of C and D operation names
        end
      end
    end
  end
end
end
end

```

Figure 31: Algorithm for Intersection Set

names and operation names. If those counts were not equal to the number of domain members, then the names were printed out. An example of this is shown in figure 33. The actual code is found in Appendix C in the routine named `noncommon.enter`.

The other procedural components written for this method were to extract data from an entry to build the object/operation sets, to propagate the attributes around the form tree and to store function name and domain member name into appropriate entries.

VIMETH-FORM-5 Common Object/Operations Form-use-#[8]

object / operations:

 <spreadsheet>

 create

 <cell>

 lock

 change value

 <row>

 insert

 <column>

 insert

Figure 32: Example of Common List Filled Form

VIMETH-FORM-6 Noncommon Object/Operations Form-use-#[21]

member / object / operations:

SAP3

⟨region⟩
color
lock
⟨column⟩
sort

SAP2

⟨region⟩
color
⟨cell⟩
absolute value

SAP1

⟨cell⟩
absolute value
⟨column⟩
sort

Figure 33: Example of Noncommon List Filled Form

Chapter 9

Conclusions and Future Work

The goal of this dissertation was to define and implement a model for a tuner tool to be used in TRIAD. This tool went beyond what other parser generator like tools had done. Several things are expected of the tuner, some of which are similar to what is expected of a parser generator.

- The tuner has to understand the model used to represent methods. This model is an attributed grammar form. A parser generator also has to understand the model used to represent programming languages. This model is often an attributed grammar.
- The tuner has to manipulate the method representation and the method instantiation and keep them both consistent. There is no way that a parser generator can manipulate the language representation and have it reflected in the programs using that language. For one thing the language representation is not maintained in the program. If the language representation is changed by the parser generator, a new compiler has to be generated and the program run through the new compiler. This is a three step operation compared to the one step operation of the tuner.

- The tuner has to work in an interactive dynamic mode. This allows the user of the method to make changes which are appropriate to the project under development and see the changes immediately. To have the same concept work with a parser generator would mean that we could tune a programming language to the particular program being developed. In essence we could change a feature of a programming language to optimize the problem being solved. Instead of having a hundred variations of a programming language, each emphasizing a particular concept of programming language theory (along with a compiler for each of these languages), we could have a few classes of languages and a compiler for each class of languages. Then the user could tune the programming language to reflect the experience accumulated and this could be propagated to the compiler with the user having to write a new compiler.

The tuner tool could be made more powerful when the following issues have been address and solved:

1. What are some common property of methods and how do they relate to properties of programming languages?
2. How can we define multiple display interfaces which display pieces of the project information base that are not related by a concept production rule?
3. How do we define data structures for attributes so that both TRIAD and the procedural components understand them?

4. How can we develop a compiler for procedural components and still take care of late binding of the attributes?
5. What data base model will efficiently store and retrieve the tree structure with the variable length data stored in it?
6. Is there a way to build generic procedural components that can have parameters other than attributes passed to them?
7. How should the attributed grammar form be modeled where the attributes and procedural components are not a one-to-one transformation?

These issues are described in the following sections to specify some future research direction.

9.1. Common Properties of Methods

Tennent in his book "Principles of Programming Languages" [105] gives a detailed summary of the properties of programming languages. He has addressed the following issues of programming languages:

1. the syntactic structure of the language,
2. the data representation, storage, and binding,
3. the control flow of programming structures, and
4. the levels of abstraction used in the language.

A similar study needs to be done with methods. In this dissertation we have developed a tuner tool that uses some of the syntactic structure of methods. The tool could be even more effective when a more detailed

study of the syntactic structure of existing methods is performed and the structure is broken up into structure classes like what has been done with programming languages. In comparison, the tuner is like the first compiler/assembler developed for programming languages.

Methods have been around for years. People have broken up methods into domain classes, but no study has been made of the common elements in methods and what are good ways of describing methods. In finding these common elements, a level of abstraction needs to be developed for methods.

One of the inherent problems with methods is that they are imprecisely defined. The same problem was present with semantics not too long ago. Several models for defining semantics were developed which has made the definition of semantics more precise. No single model for semantics has taken hold as the best model. Each model is powerful enough to capture a certain aspect of semantic definition. We have presented one model for defining methods which is closer to a syntax model than a semantic model. Some other models need to be developed which capture more of the semantic meaning in methods. These new models will be more effective if we can characterize the properties of methods.

9.2. Multiple Display Interfaces

Any display interface can show the project information according to the concept productions rules. This is the typical way that project information is built and displayed. What is more challenging is to display the information according to a user profile. For example, the documentation clerk might be only interested in that portion of the project data base where the manuals are developed from. Documentation

is normally intermixed with the code, management information, and the test data and result. The documentation clerk is not interested in viewing all this other information while he is browsing through the documentation. A user profile and an accompanying display interface needs to be developed so that the clerk would view that portion of the project information of interest to him.

The issue is how to define this interface and allow the tuner to manipulate it. If new concept production rules are added to a method, how does the tuner decide what affect these rules have on each of the display interfaces.

Another issue is when to display the tags and when not to. At some point in the project information data base, the display of the tags may clutter up the screen. For example, if we are in the section of the grammar form that defines a grammar for a particular programming language, at what point do we just print the input source code without the tags. Do we do this at a module level or do we do it at the statement level? How do we define the interface if we are not going to display the tags? Is there a time when we want to sometimes display the tags and other times not display the tags?

9.3. Data Structures For Attributes

The only data structures that are currently available in TRIAD are integer, string, and a list of integers. They were powerful enough to handle most of the attributes for the examples that we used. However, we did reach our limitation in the virtual interface method example. Each of the three types of attributes were hard wired into TRIAD. If each new attribute type has to be hard wired into TRIAD, the power of

TRIAD will be limited. Therefore there needs to be a way of describing data structures to TRIAD. This capability should also be added to the lisp like language as the language is going to need to access elements of the data structure.

ALOE only allows tree structures as data structures for attributes. The same tree traversal routines used in traversing abstract syntax trees can be used to access attributes in TRIAD. This allows for a simple interface, but not all data structures are easy to visualize as trees. For example, doubly linked lists can not be represented as trees, nor can any network type data structure.

The primary elements of a data structure should be integer, real, string, and pointer. There are special problems with pointers. In programs that use pointers, the pointers are given a relative address that is updated when the program is loaded. This is not so with attributes. Because of the dynamic nature of the nodes with which attributes are associated, there is no guarantee that the nodes and its associated attributes will be loaded in the same relative location each time the node is accessed. Each node has to be uniquely identified. A table then has to be set up with the unique node id and its relative location in memory. Then to resolve a pointer you use the node id stored in the pointer and, using the table, find the node. If the pointer attribute points to some other entity than a node (like another attribute), then there must be something in that entity that will uniquely identify itself.

9.4. Compiler For Procedural Components

We have built a lisp like language with an associated interpreter. An interpreter is easy to write and to expand, but it is inherently slow. The amount of overhead associated with processing each command is generally more than the actual execution. If the procedural components are small, then this overhead is not noticed. However, if the procedural components are, for instance... "find all the common objects and operation for the virtual interface method", then the delay is more than what the user finds tolerable. In one example of using the virtual interface method to define the spreadsheet domain interface, it took about 10 minutes to calculate the commons.

To overcome this problem, the procedural components need to be compiled. There are two opportunities to compile the procedural components, when they are defined, or when they are read in. If delayed until read in, then only the source code need to be maintained with the method and no linking of the program is needed. If compiled when defined, then a linker needs to be developed and both the source code and the object code needs to be maintained with the method.

Another problem associated with a compiled version is the linking of the attributes at execute time. The attributes would be similar to input data to a normal program. Reading in attributes should be transparent to the user. There is also the problem that the attributes at one node which uses this procedural component might not have the attributes in the same order as another node which uses this procedural component. Reading in data is generally done sequentially. The same problem is associated with saving the new values of the attributes. This corresponds to outputting data which again needs to be transparent to the user.

9.5. Data Base Model

Supporting the voluminous information developed during all phases of a project requires the use of a data base management system. A relational data base system has the ability to model many kinds of information and has a powerful query language which allows easy access to the information data base [107]. The use of a relational data base has the following advantages for TRIAD:

- efficient handling of large volumes of information,
- ability to model the attributed grammar form underlying the project information generated by a method,
- efficient query language that extracts information using the concept symbols of the method,
- some generic global semantic functions (i.e. aggregate operators) like average, total, number, etc., and
- ability to dynamically add new relations to the data base for when new attributes and symbols are added to a method.

There are drawbacks to using a relational data base. Some of these are:

- There has not been found an efficient way to store variable length data. Several individuals have addressed the issue for different domains, but no general solution has yet been proposed [79, 69, 70, 34].

- The query language is designed to only handle unanticipated factual queries. Although the aggregate operators present the ability to answer some deductive questions, these are limited to some simple operations like find the average or get the maximum value.
- Mapping the tree structure of the attributed grammar form to the relational model does not take into consideration the anticipation of the next piece of information for efficient management of memory. The constraint mechanism of the attributed grammar form can not be enforced by the relational model.

Another possible data base model that could be used is the entity relationship model. Some research has been done for the software engineering and CAD/CAM domains [76, 75, 53, 62]. A study of how this could be used in a meta environment still needs to be done.

9.6. Generic Procedural Components

In coding the procedural components for the three test cases it became apparent that certain procedural components were repeating themselves with the only difference being the name of an attribute. This lead to asking some interesting questions, like:

- can we have a procedural component that has the same form as a program, where the program has access to a library of functions?
- can we have procedural components that pass parameters?

AD-A158 102

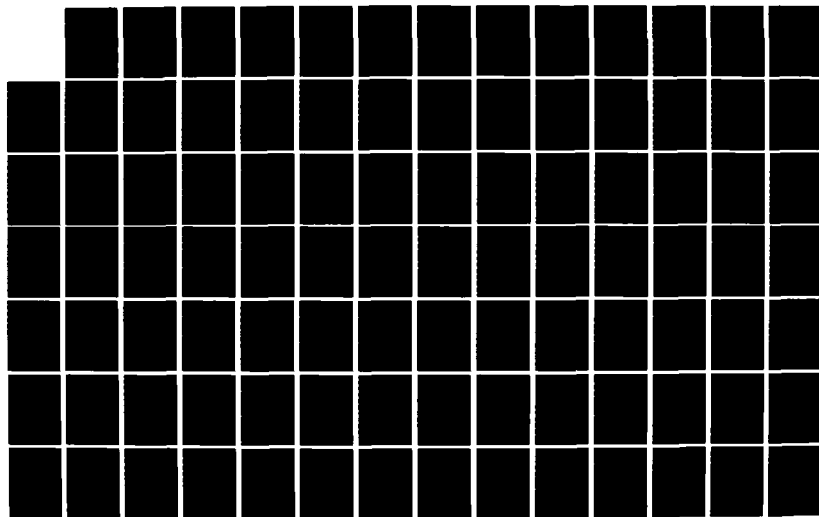
A META SYSTEM FOR GENERATING SOFTWARE ENGINEERING
ENVIRONMENTS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON
AFB OH W L MCKNIGHT 1985 AFIT/CI/NR-85-710

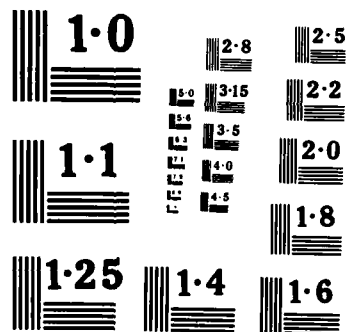
3/1

UNCLASSIFIED

F/G 9/2

NL





NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

- can we have macro type procedural components?

One wonders why these questions have not been asked earlier, since the concept of procedural components have been around for some time.

Another interesting question that needs answering is where we draw the line between developing a procedural component to do a function and developing an interface to a tool to do that function. This is an instrumentation issue that will take some experience with procedural components before it can be answered.

9.7. More General Attributed Grammar Form

Our current definition of attributed grammar form is not attractive enough for the following reasons:

- The names and types of attributes associated with the symbols in the form grammar may not make sense when associated with the interpreted symbols. For example, the value of currency may be expressed in different denominations in different countries.
- Just as symbols in the original grammar represents a family of symbols, we would like an attribute to represent a family of attributes and a semantic rule to represent a family of semantic rules. This way, many similar attributes and semantic rules could be added without essentially changing the method of semantic evaluation.
- Quite often, interpreted grammar symbols are, in some sense, a specialization of the original grammar symbol. We would

like to restrict the values some attributes of the interpreted symbol are allowed to have based on this specialization.

Bibliography

1. Aho, A. V. and J. D. Ullman. Automatic Computations. Volume I and II: The Theory of Parsing, Translation, and Compiling. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.
2. Ambriola, V., G. E. Kaiser, and R. J. Ellison. An Action Routine Model for ALOE. Technical Report CMU-CS-84-158, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August, 1984.
3. Archer, J. E. Jr. and R. Conway. COPE: A Cooperative Programming Environment. Technical Report TR 81-459, Department of Computer Science, Cornell University, Ithaca, New York, 14853, June, 1981.
4. Ashok, V., W. L. McKnight, and J. Ramanathan. Integrated Environment For Information Management In VLSI Design. Proceedings of the Computer Data Engineering Conference, IEEE Computer Society, Los Angeles, California, April, 1984, pp. 12-19.
5. Ashok, V., W. L. McKnight, and J. Ramanathan. Uniform Support for Information Handling and Problem Solving Required by the VLSI Design Process. ACM IEEE 21st Design Automation Conference, IEEE Computer Society, Albuquerque, New Mexico, June, 1984, pp. 694-695.
6. Beetem, J. and A. Nigam. VLSI Design Using Sheets and Types. Technical Report RC 9854, IBM Thomas J. Watson Research Center, Yorktown Heights. New York, February, 1983.
7. Blattner, M. The Decidability of the Equivalence of Context-free Grammar Forms. 20th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Puerto Rico, October, 1979, pp. 91-96.
8. Bochmann, G. V. Attribute Grammars and Compilation: Program Evaluation in Several Phases. 54. University of Montreal, Montreal, Canada, August, 1974.

9. Bochmann, G. V. Semantic Attributes for Grammars with Regular Expressions. 195, University of Montreal, Montreal, Canada, 1975.
10. Bochmann, G. V. "Semantic Evaluation From Left to Right". Communications of the ACM 19, 2 (February 1976), 55-62.
11. Campbell, R. H. and P. C. Richards. SAGA: A System to Automata the Management of Software Productions. Technical Report UIUCDCS-R-81-1048, Department of Computer Science, University of Illinois at Urbana-Champaign, January, 1981.
12. Campbell, R. H. and P. A. Kirsliis. The SAGA Project: A System for Software Development. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, April, 1984, pp. 73-80.
13. Canepa, M., E. Weber, and H. Talley. "VLSI in FOCUS: Designing a 32-bit CPU Chip". VLSI Design, (January/February 1983), 20-24.
14. Chesi, M., E. Dameri, M. P. Franceschi, M. G. Gatti, and C. Simonelli. ISDE: An Interactive Software Development Environment. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, April, 1984, pp. 81-88.
15. Cohen, R. and E. Harry. Automatic Generation of Near-Optimal Linear-time Translators for Non-Circular Attribute Grammars. Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, ACM, San Antonio, Texas, January, 1979, pp. 121-134.
16. Coutaz, J. The Box, A Layout Abstraction For User Interface Toolkits. Technical Report CMU-CS-84-167, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, December, 1984.
17. Cremers, A. and S. Ginsburg. "The Structure of Context-Free Grammatical Families". Journal of Computer and System Sciences 15, 3 (December 1977), 262-279.
18. Davis, C. and C. Vick. "The Software Development System". IEEE Transactions on Software Engineering SE-3, 1 (January 1977), 69-84.

19. Delisle, N. M., D. E. Menicosy, and M. D. Schwartz. Viewing a Programming Environment as a Single Tool. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, Pittsburgh, Pennsylvania, April, 1984, pp. 49-56.
20. Demers, A., T. Reps, and T. Teitelbaum. Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors. *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, ACM, Williamsburg, Virginia, January, 1981, pp. 105-116.
21. Denning, P. J., J. B. Dennis, and J. E. Qualitz. Machines, Languages, and Computation. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.
22. DeRemer, F. and H. H. Kron. "Programming-in-the-Large Versus Programming-in-the-Small". *IEEE Transactions on Software Engineering* SE-2 (June 1976), 80-86.
23. Deutsch, L. P. and E. A. Taft. Requirements for an Experimental Programming Environment. CSL-80-10, Xerox Palo Alto Research Center, Palo Alto, California, June, 1980.
24. Dobbs, V. and S. A. Mamrak. A Methodology for the Design and Implementation of Virtual Interfaces.
25. Dobbs, V. An Automated Methodology for the Design and Implementation of Virtual Interfaces. Ph.D. Th., The Ohio State University, Columbus, Ohio, July 1985.
26. Donzeau-Gouge, V., G. Huet, G. Kahn, and B. Lang. Programming Environments Based on Structured Editors: the Mentor Experience. Inria, May, 1980.
27. Farrow, R. Experience With an Attribute Grammar-Based Compiler. *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM, Albuquerque, New Mexico, January, 1982, pp. 95-107.
28. Farrow, R. LINGUIST-86 Yet Another Translator Writing System Based On Attribute Grammars. *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, ACM, Boston, Massachusetts, June, 1982, pp. 160-171.

- 29.** Farrow, R. Sub-Protocol-Evaluators for Attribute Grammars. Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, ACM, Montreal, Canada, June, 1984, pp. 70-80.
- 30.** Feiler, P. H. and R. Medina-Mora. An Incremental Programming Environment. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April, 1980.
- 31.** Feiler, P. H. and R. Medina-Mora. "An Incremental Programming Environment". IEEE Transactions on Software Engineering SE-7, 7 (September 1981), 472-481.
- 32.** Fischer, C. N., G. F. Johnson, J. Mauney, A. Pal, and D. L. Stock. The Poe Language-Based Editor Project. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, April, 1984, pp. 21-29.
- 33.** Tutorial on Software Design Techniques, Long Beach, California, 1980.
- 34.** Gajski, D., W. Kim, and S. Fushimi. A Parallel Pipelined Relational Query Processor: An Architectural Overview. Report RJ 4087, IBM Research Division, San Jose, California, October, 1983.
- 35.** Ganzinger, H., R. Giegerich, U. Moncke, and R. Wilhelm. A Truly Generative Semantic-Directed Compiler Generator. Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, ACM, Boston, Massachusetts, June, 1982, pp. 172-184.
- 36.** Garlan, D. B. and P. L. Miller. GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, April, 1984, pp. 65-72.
- 37.** Geschke, C. M., J. H. Morris Jr., and E. H. Satterwaite. "Early Experience with MESA". Communication of the ACM 20, 8 (August 1977), 540-553.
- 38.** Giacalone, A., M. C. Rinard, and T. W. Doepfner Jr. IDEOSY An Ideographic and Interactive Program Description System. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, April, 1984, pp. 15-20.

39. Goates, G. B., T. R. Harris, R. E. Oettel, and H. M. Waldron III. "Storage/Logic Array Design: Reducing Theory to Practice". VLSI Design 3, 4 (July/August 1982), 56-62.
40. Goldberg, A. The Influence of an Object-oriented Language on the Programming Environment. Proceedings of the ACM Computer Science Conference, ACM, Orlando, Florida, February, 1983, pp. 35-54.
41. Habermann, A. N. An Overview of the Gandalf Project. Department of Computer Science, Carnegie-Mellon, Pittsburgh, Pennsylvania, 1979.
42. Habermann, A. N. The Gandalf Project. Department of Computer Science, Carnegie-Mellon, Pittsburgh, Pennsylvania, 1981.
43. Habermann, A. N. and D. S. Notkin. The Gandalf Software Development Environment. Department of Computer Science, Carnegie-Mellon, Pittsburgh, Pennsylvania, 1982.
44. Hayes, P. J. Executable Interface Definitions using Form-Based Interface Abstraction. Technical Report CMU-CS-84-110, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March, 1984.
45. Hopcroft, J. E. and J. D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Co., Reading, Massachusetts, 1979.
46. Horgan, J. R. and D. J. Moore. Techniques for Improving Language-Based Editors. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, April, 1984, pp. 7-14.
47. Howden, W. "DISSECT - A Symbolic Evaluation and Program Testing System". IEEE Transactions on Software Engineering SE-4, 1 (January 1978). 70-73.
48. Huang, K. T. and C. C. Wang. Form Design by Form-Filling. Report RC 10805, IBM T. J. Watson Research Center, Yorktown Heights, New York, August, 1984.
49. Jackson, M. A.. Principles of Program Design. Academic Press, New York, 1975.

50. Jazayeri, M. and K. G. Walter. Alternating Semantic Evaluator. Proceedings of the Annual Conference, ACM, Minneapolis, Minnesota, October, 1975, pp. 230-234.
51. Jazayeri, M., W. F. Ogden, and W. C. Rounds. On the Complexity of the Circularity Test for Attribute Grammars. Conference Record of the Second ACM Symposium on Principles of Programming Languages, ACM, Palo Alto, California, January, 1975, pp. 119-129.
52. Jazayeri, M. and D. Pozefsky. "Space-Efficient Storage Management in an Attribute Grammar Evaluator". ACM Transactions on Programming Languages and Systems 3, 4 (October 1981), 388-404.
53. Johnson, H. R., J. E. Schweitzer, and E. R. Warkentine. A DBMS Facility for Handling Structured Engineering Entities. Proceedings of Annual Meeting of Engineering Design Applications, IEEE, San Jose, California, May, 1983, pp. 3-11.
54. Jordal, M. The Architecture of a Meta Environment for Supporting Methods. Master Th., The Ohio State University, Columbus, Ohio, May 1985.
55. Jourdan, M. Strongly Non-Circular Attribute Grammars and Their Recursive Evaluation. Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM, Montreal, Canada, June, 1984, pp. 81-93.
56. Jullig, R. K. and F. DeRemer. Regular Right-Part Attribute Grammars. Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM, Montreal, Canada, June, 1984, pp. 171-179.
57. Kiper, J. and J. Ramanathan. An Approach to Human Engineering Existing Compilers and Other Tools. Triad-84-5, The Ohio State University, Columbus, Ohio, November, 1984.
58. Knuth, D. E. "Semantics of Context-Free Languages". Mathematical Systems Theory 2, 1 (1968), 127-145.
59. Knuth, D. E. "Semantics of Context-Free Languages: Correction". Mathematical Systems Theory 5, 1 (1971), 95-96.
60. Kuo, J. C., C. Li, and J. Ramanathan. A Form-Based Approach to Human Engineering Methodologies. Proceeding of the 6th International Conference on Software Engineering. IEEE Computer Society, Tokyo, Japan, September, 1982, pp. 254-263.

61. Kuo, J. C. Design and Implementation of a Form-Based Software Environment. Ph.D. Th., The Ohio State University, Columbus, Ohio, August 1983.
62. Kutay, A. R. and C. M. Eastman. Transaction Management in Engineering Databases. Proceedings of Annual Meeting of Engineering Design Applications, IEEE, San Jose, California, May, 1983, pp. 37-80.
63. Lewis, H. R. and C. H. Papadimitriou. Elements of the Theory of Computation. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
64. Li, C. H. Extensions to the Attribute Grammar Form Model to Model Meta Software Engineering Environments. Ph.D. Th., The Ohio State University, Columbus, Ohio, February 1985.
65. Linton, M. A. Queries and Views of Programs Using a Relational Database System. Ph.D. Th., University of California, Berkeley, California, December 1983.
66. Linton, M. A. Implementing Relational Views of Programs. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, April, 1984, pp. 132-140.
67. Liskov, B. and S. Zilles. An Introduction to Formal Specifications of Data Abstraction. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.
68. Logrippo, L. and D. R. Skuce. "File Structures, Program Structures, and Attribute Grammars". IEEE Transactions on Software Engineering May, 3 (SE-9 1983), 260-266.
69. Lorie, R. and W. Plouffe. Relational Databases for Engineering Data. Research Report RJ 3847, IBM Research Laboratory, San Jose, California, April, 1983.
70. Lorie, R. and W. Plouffe. Complex Objects and Their Use in Design Transactions. Proceedings of Annual Meeting of Engineering Design Applications, IEEE, San Jose, California, May, 1983, pp. 115-121.
71. Maurer, H. A., A. Solomaa, and D. Wood. Strict Context-Free Grammar Forms: Completeness and Decidability. 78-CS-19, McMaster University, 1978.

- 72.** McKnight, W. L. and J. Ramanathan. A Meta System For Generating Software Engineering Environments.
- 73.** Mead, C. and L. Conway. Introduction to VLSI Systems. Addison-Wesley Publishing Co., Reading, Massachusetts, 1980.
- 74.** Medina-Mora, R., D. S. Notkin, and R. J. Ellison. ALOE Users' and Implementors' Guide. Second edition, Department of Computer Science, Pittsburgh, Pennsylvania, 1982.
- 75.** Neumann, T. On Representing the Design Information in a Common Database. Proceedings of Annual Meeting of Engineering Design Applications, IEEE, San Jose, California, May, 1983, pp. 81-87.
- 76.** Olumi, M., G. Wiederhold, C. Hauser, P. Lucas, and J. Mehl. Software Project Databases. Research Report RJ 3862, IBM San Jose Research Laboratory, San Jose, California, April, 1983.
- 77.** Osterweil, L. J. and L. D. Fosdick. "DAVE - A Validation Error Detection and Documentation System for Fortran Programs". Software - Practice and Experience 6, 4 (October - December 1976), 473-486.
- 78.** Ousterhout, J. K. "Caesar: An Interactive Editor for VLSI Layouts". VLSI Design I (Fourth Quarter 1981), .
- 79.** Powell, M. L. and M. A. Linton. Database Support for Programming Environments. Proceedings of Annual Meeting of Engineering Design Applications, IEEE, San Jose, California, May, 1983, pp. 63-70.
- 80.** Raiha, K. On Attribute Grammars and Their Use in a Compiler Writing System. Report A-1977-4, Department of Computer Science, University of Helsinki, Helsinki, Finland, August, 1977.
- 81.** Ramanathan, J. and D. Soni. "Design and Implementation of an Adaptable Software Environment". Computer Languages 8, 3/4 (1983), 139-159.
- 82.** Reiss, S. P. PECAN: Program Development Systems That Supports Multiple Views. 7th International Conference on Software Engineering. IEEE Computer Society, Orlando, Florida, March, 1984. pp. 324-333.

83. Reiss, S. P. Graphical Program Development with PECAN Program Development Systems. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania. April, 1984, pp. 30-41.
84. Reps, T. W. Generating Language-Based Environments. Ph.D. Th., Cornell University, Ithaca, New York, August 1982.
85. Reps, T. W. and T. Teitelbaum. "The Synthesizer Generator". SIGPLAN Notices 19, 5 (May 1984), 42-48.
86. Roberts, R. B. and I. P. Goldstein. The FRL Manual. MIT Artificial Intelligence Laboratory, 1977.
87. Ross, D. T. and K. E. Schoman. "Structured Analysis for Requirements Definition". IEEE Transactions on Software Engineering SE-3, 1 (January 1977), 6-15.
88. Ross, D. T. Structured Analysis(SA): A Language for Communicating Ideas. In Tutorial on Software Design Techniques, Peter Freeman and Anthony I. Wasserman, Eds., IEEE Computer Society, 1980, pp. 107-125.
89. Schwartz, M. D., N. M. Delisle, and V. S. Begwani. Incremental Compilation in Magpie. Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM, Montreal, Canada, June, 1984, pp. 122-131.
90. Shu, N. C., V. Y. Lum, F. C. Tung, and C. L. Chang. "Specification of Forms Processing and Business Procedures for Office Automation". IEEE Transactions on Software Engineering SE-8, 5 (September 1982), 499-512.
91. Shubra, C. and J. Ramanathan. Template Based Software Design and Manufacture.
92. Shubra, C. J. Jr. The Identification of Semantics For the File DataBase Problem Domain and Their Use in Template-Based Environment. Ph.D. Th., The Ohio State University, Columbus, Ohio, August 1984.

93. Skedzeleski, S. K. Defintion and Use of Attribute Reevaluation in Attributed Grammars. Ph.D. Th., University of Wisconsin-Madison, Madison, Wisconsin, October 1978.
94. The Smalltalk80 System: A User Guide and Reference Manual. Xerox Palo Alto Research Center, Palo Alto, California, 1982.
95. Sobol, R. "The Universal Synchronous Machine". VLSI Design 4, 7 (November 1983), 60-66.
96. Soni, D. A. Design and Modeling of TRIAD - An Adaptable, Integrated Software Environment. Ph.D. Th., The Ohio State University, Columbus, Ohio, June 1983.
97. Standish, T. A. A Preliminary Philosophy for ARCTURUS. Computer Science Department, University of California, Irvine, California, 1980.
98. Standish, T. A. and R. N. Taylor. Arcturus: a Prototype Advanced Ada Prgoramming Environment. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, April, 1984, pp. 57-64.
99. Stay, J. F. HIPO and Interactive Program Design. In Tutorial on Software Design Techniqes, Peter Freeman and Anthony I. Wasserman, Eds., IEEE Computer Society, 1980, pp. 253-257.
100. Stoll, P. A. "How to Avoid Synchronization Problems". VLSI Design 3, 6 (November/December 1982), 56-59.
101. Teichroew, D. and E. A. Hershey, III. "PSL, PSA: A Computer-Aided Technique for Structured Documentation and Analysis for Information Processing Systems". IEEE Transactions on Software Engineering SE-3, 1 (January 1977), 41-48.
102. Teitelbaum, T. and T. W. Reps. The Cornell Synthesizer: A Syntax-Directed Programming Environment. Cornell University, 1980.
103. Teitelbaum, T., T. W. Reps, and S. Hortwitz. "The Why and Wherefore of the Cornell Program Synthesizer". SIGPLAN Notices 16, 6 (June 1981), 8-16.

```

      {setq count { + 1 count } } } } }
{setq attribs { add-item-to-list attribs attribute.node-number } }

```

```

type.update
{ char sstring }
{setq sstring { buffer-to-string } }
{setq attribute.type 0 }
{ if { string-compare sstring "int" }
  {setq attribute.type 1 } }
{ if { string-compare sstring "str" }
  {setq attribute.type 2 } }
{ if { string-compare sstring "list" }
  {setq attribute.type 3 } }
{ if { ! attribute.type }
{ begin
  { message "Legal values are int/str/list" }
  { quit } } } }

```

```

default-value.update
{ char sstring }
{setq sstring { buffer-to-string } }
{ if { & { = 1 attribute.type } { ! { integer-test sstring } } }
{ begin
  { message "This value must be an integer" }
  { quit } } }
{setq attribute.value sstring }

```

```

action.delete
{setq actions { delete-item-from-list actions { node-number } } }

```

```

action.enter
{setq action.node-number { node-number } }

```

```

attribute.delete
{setq attribs { delete-item-from-list attribs { node-number } } }

```

```

{ if { string-compare cstring "I" }
  { setq attribute-set-form.class 2 } }
{ if { string-compare cstring "G" }
  { setq attribute-set-form.class 3 } }

```

attribute-set.enter

```
{ setq attribute-set.class attribute-set-form.class }
```

attribute.enter

```

{ setq attribute.class attribute-set.class }
{ setq attribute.node-number { node-number } }

```

attribute-name.update

```

{ char sstring }
{ char bname }
{ int llen }
{ int count }
{ int anode }
{ setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{ setq attribute.aname sstring }
{ setq llen { size-of-list attribs } }
{ if { != 0 llen }
{ begin
  { setq count 1 }
  { while { = count llen }
  { begin
    { setq anode { get-next-item-from-list attribs count } }
    { if { != anode attribute.node-number }
    { begin
      { setq bname { get-attrib-at-node anode "aname" } }
      { if { string-compare sstring bname }
      { begin
        { message "This attribute name is already defined" }
        { quit } } } } } }

```


method-name.visit

```
{ if { string-compare method-name "" }
{ begin
  { message "Methodology Name must be specified first" }
  { quit } } }
```

action-procedure.update

```
{setq action-procedure.define 0 }
{ if { != 0 { length { buffer-to-string } } }
  {setq action-procedure.define 1 } }
```

action.visit

```
{ if { != 0 { length action-name.aname } }
{ begin
  { if { = 0 action-procedure.define }
    { message "Action procedure not yet defined" } } } }
{setq action.aname action-name.aname }
```

action-form.enter

```
{ if { = 0 { node-child-form } }
{ begin
  { erase-buffer }
  { insert-string "      Form-use-#" }
  { set-dot 17 } }
{ begin
  { erase-buffer }
  { insert-string action.aname }
  { insert-string "      Form-use-#" }
  { insert-string { int-to-string { node-child-form } } }
  { insert-string " " }
  { set-dot { buffer-size } } } }
```

attribute-set-form.enter

```
{ char cstring }
{setq cstring { substr { buffer-to-string } 1 1 } }
{ if { string-compare cstring "S" }
  {setq attribute-set-form.class 1 } }
```

```

form-number.update
{ char sstring }
{ char bname }
{ int llen }
{ int count }
{ int fvalue }
{ int bvalue }
{ int anode }
{setq sstring { buffer-to-string } }
{ if { ! { integer-test sstring } }
{ begin
  { message "This entry must contain an integer" }
  { quit } } }
{setq fvalue { string-to-int sstring } }
{ if { = 0 fvalue }
{ begin
  { message "Form number of 0 is not allowed" }
  { quit } } }
{setq llen { size-of-list bforms } }
{ if { != 0 llen }
{ begin
  {setq count 1 }
  { while { <= count llen }
  { begin
    {setq anode { get-next-item-from-list bforms count } }
    { if { != anode form-definition.node-number }
    { begin
      {setq bvalue { get-attr-at-node anode "value" } }
      { if { = bvalue fvalue }
      { begin
        { message "This form number already used" }
        { quit } } } } }
      {setq count { + 1 count } } } } } }
  {setq bforms { add-item-to-list bforms form-definition.node-number } }
  {setq form-definition.value fvalue }
}
}

grammar-name.enter
{ erase-buffer }
{ insert-string grammar-name }

```

```

{ begin
  { setq count 1 }
  { while { <= count llen }
    { begin
      { setq anode { get-next-item-from-list actions count } }
      { if { != anode action.node-number }
        { begin
          { setq bname { get-attr-at-node anode "aname" } }
          { if { string-compare sstring bname }
            { begin
              { message "This action name already used" }
              { quit } } } }
          { setq count { + 1 count } } } } } }
    { setq actions { add-item-to-list actions action.node-number } }
  }

```

method-name.update

```

{ char name }
{ char subname }
{ int sdot }
{ int pos }
{ int slen }
{ setq name { buffer-to-string } }
{ setq sdot { string-to-char "." } }
{ setq pos { find-char-in-string name sdot } }
{ if { = pos 0 }
  { begin
    { message "This entry does not have an extension" }
    { quit } } }
{ setq slen { length name } }
{ setq subname { substr name pos { + 1 { - slen pos } } } }
{ if { ! { string-compare subname ".md" } }
  { begin
    { message "This entry does not have .md extension" }
    { quit } } }
{ setq method-name name }
{ setq subname { substr name 1 { - pos 1 } } }
{ setq grammar-name { concat subname ".grm" } }
{ setq method-interp { concat subname "-FORM-" } }

```

Finally, we present the set of procedural components that was implemented. The first part of each name indicates the symbol to which the procedural component is tied to. The last part of each name indicates under which condition this procedural component will be fired.

```

action-name.update
{ char sstring }
{ char sbstring }
{ char bname }
{ int slen }
{ int pos }
{ int sdot }
{ int llen }
{ int count }
{ int anode }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{ if { = pos 0 }
{ begin
  { message "This entry does not have an extension" }
  { quit } } }
{setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{ if { & { ! { string-compare sbstring ".visit" } }
  { & { ! { string-compare sbstring ".enter" } }
  { & { ! { string-compare sbstring ".update" } }
  { & { ! { string-compare sbstring ".repeat" } }
  { & { ! { string-compare sbstring ".delete" } }
  { ! { string-compare sbstring ".create" } } } } } } }
{ begin
  { message "This entry does not have a proper extension" }
  { quit } } }
{setq action-name.aname sstring }
{setq llen { size-of-list actions } }
{ if { != 0 llen }

```

Next, we will show the list of attribute types that were used in the meta method. These attribute types are assigned to the grammar symbols.

Name	Class	Type
aname	Synth	str
symbols	Global	list
products	Global	list
attribs	Global	list
actions	Global	list
ter-symbols	Global	list
bforms	Global	list
method-name	Global	str
method-interp	Global	str
grammar-name	Global	str
method-help	Global	str
start-form	Global	int
start-symbol	Global	str
define	Synth	int
class	Inherit	int
node-number	Inherit	int
type	Synth	int
value	Synth	str
start	Inherit	int
node	Synth	int

15. $\langle \text{production-rule-form} \rangle ::= \langle \text{production-rule} \rangle$
16. $\langle \text{action} \rangle ::= \langle \text{action-name} \rangle$
 $\langle \text{action-procedure} \rangle$
17. $\langle \text{attribute-set} \rangle ::= \langle \text{attribute} \rangle^*$
18. $\langle \text{attribute} \rangle ::= \langle \text{attribute-name} \rangle$
 $\langle \text{type} \rangle$
 $\langle \text{default-value} \rangle$
19. $\langle \text{symbol} \rangle ::= \langle \text{symbol-name} \rangle$
 $\langle \text{sattribute} \rangle^*$
 $\langle \text{saction-name} \rangle^*$
20. $\langle \text{sattribute} \rangle ::= \langle \text{sattribute-name} \rangle$
 $\langle \text{sdefault-value} \rangle$
21. $\langle \text{production-rule} \rangle ::= \langle \text{production-number} \rangle$
 $\langle \text{left-hand-symbol} \rangle$
 $\langle \text{right-hand-symbol} \rangle^+$
 $\langle \text{semantic-functions} \rangle$
22. $\langle \text{right-hand-symbol} \rangle ::= \langle \text{handle-name} \rangle$
 $\langle \text{operation} \rangle$
 $\langle \text{form-representation} \rangle$
23. $\langle \text{operation} \rangle ::= \langle \text{kleene-star} \rangle$
 $\langle \text{plus} \rangle$
 $\langle \text{zero-one} \rangle$
24. $\langle \text{form-representation} \rangle ::= \langle \text{heading} \rangle$
 $\langle \text{displayable-help-information} \rangle$
 $\langle \text{prefilled-entry} \rangle$
 $\langle \text{entry-updatable} \rangle$
 $\langle \text{number-of-lines} \rangle$
25. $\langle \text{semantic-functions} \rangle ::= \langle \text{saction-name} \rangle^*$

1. $\langle \text{methodology} \rangle ::= \langle \text{methodology-name} \rangle$
 $\langle \text{grammar-definition-form} \rangle$
 $\langle \text{form-definition} \rangle^+$
2. $\langle \text{grammar-definition-form} \rangle ::= \langle \text{grammar-definition} \rangle$
3. $\langle \text{form-definition} \rangle ::= \langle \text{form-number} \rangle$
 $\langle \text{root-number} \rangle$
4. $\langle \text{grammar-definition} \rangle ::= \langle \text{grammar-name} \rangle$
 $\langle \text{action-set} \rangle$
 $\langle \text{attributes} \rangle$
 $\langle \text{symbols} \rangle$
 $\langle \text{production-set} \rangle$
5. $\langle \text{action-set} \rangle ::= \langle \text{action-form} \rangle^*$
6. $\langle \text{action-form} \rangle ::= \langle \text{action} \rangle$
7. $\langle \text{attributes} \rangle ::= \langle \text{attribute-set-form} \rangle$
 $\langle \text{attribute-set-form} \rangle$
 $\langle \text{attribute-set-form} \rangle$
8. $\langle \text{attribute-set-form} \rangle ::= \langle \text{attribute-set} \rangle$
9. $\langle \text{symbols} \rangle ::= \langle \text{start-symbol} \rangle$
 $\langle \text{terminal-symbol} \rangle$
 $\langle \text{non-terminal-symbol} \rangle$
10. $\langle \text{start-symbol} \rangle ::= \langle \text{symbol-form} \rangle$
 $\langle \text{heading} \rangle$
 $\langle \text{displayable-help-information} \rangle$
11. $\langle \text{symbol-form} \rangle ::= \langle \text{symbol} \rangle$
12. $\langle \text{terminal-symbol} \rangle ::= \langle \text{symbol-form} \rangle^+$
13. $\langle \text{non-terminal-symbol} \rangle ::= \langle \text{symbol-form} \rangle^*$
14. $\langle \text{production-set} \rangle ::= \langle \text{production-rule-form} \rangle^+$

META-FORM-6

Production Rule

Form-use-#[]

Production Number:

Left Hand Symbol:

Right Hand Symbol [more?]: (given in left to right order)

Name:

Operation:

Kleene Star:

+:

0/1:

Form Representation:

Heading:

Displayable Help Information:

Prefilled Entry:

Entry Updatable:

Number of lines:

Semantic Functions:

Name [more?]:

Next, the list of all the grammar production rules are presented. * represents the kleene star; and + is the plus function. The productions rules are presented using grammars with right regular parts.

META-FORM-3**Action**

Form-use-#[]

Action Name:

Action Procedure:

META-FORM-4**Attribute Set**

Form-use-#[]

Attribute [more?]:

Name:

Type:

Default Value:

META-FORM-5**Symbol**

Form-use-#[]

Symbol Name:

Attributes [more?]:

Name:

Default Value:

Action Name [more?]:

META-FORM-2**Grammar Definition****Form-use-#[]****Grammar Name:****Action Set:**

{3} Action [more?]: Form-use-#[]

Attributes:

{4} Attribute Set: Synthesized Local Form-use-#[]

{4} Attribute Set: Inherited Local Form-use-#[]

{4} Attribute Set: Global Form-use-#[]

Symbols:**Start Symbol:**

{5} Symbol: Form-use-#[]

Heading:**Displayable Help Information:****Terminal Symbol:**

{5} Symbol [more?]: Form-use-#[]

Non-terminal Symbol:

{5} Symbol [more?]: Form-use-#[]

Production Set:

{6} Production Rule [more?]: Form-use-#[]

Appendix A

Implementation of the Tuner in Tuner

In this appendix, the blankforms, the attributed grammar production rules, and the procedural components for the meta method are presented. This is a small method with 6 blankforms, 25 production rules, 20 attribute types, and 46 procedural components. First, we will show the blankforms.

META-FORM-1	Methodology	Form-use-#[]
<hr/>		
Methodology Name:		
<hr/>		
{2} Grammar Definition:	Form-use-#[]	
<hr/>		
Form Definition [more?]:		
<hr/>		
Form Number:		
<hr/>		
Production Number:		
<hr/>		

104. Teitelman, W. and L. Masinter. "The Interlisp Programming Environment". Computer 14, 4 (April 1981), 25-33.
105. Tennent, R. D.. Principles of Programming Languages. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.
106. Tucker, M. and L. Scheffer. "A Constrained Design Methodology for VLSI". VLSI Design , (May/June 1982), 60-65.
107. Ullman, J. D.. Principles of Database Systems. Computer Science Press, Inc., Rockville, Maryland, 1980.
108. Wiest, J. and F. Levy. A Management Guide to PERT/CMP. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.
109. Wood, D.. Lecture Notes in Computer Science. Volume : Grammar and L-Forms: An Introduction. Springer-Verlag, New York, New York, 1980.
110. Yourdon, E. and L. L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1975.
111. Zelkowitz, M. V. A Small Contribution To Editing With a Syntax Directed Editor. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, April, 1984, pp. 1-6.

start-symbol.enter

```
{setq start-symbol.class 1 }
{setq start-symbol.start 1 }
```

terminal-symbol.enter

```
{setq terminal-symbol.class 2 }
{setq terminal-symbol.start 2 }
```

non-terminal-symbol.enter

```
{setq non-terminal-symbol.class 1 }
{setq non-terminal-symbol.start 2 }
```

symbol-form.enter

```
{setq symbol-form.class {get-attr-at-node {node-parent} "class" }}
{setq symbol-form.start {get-attr-at-node {node-parent} "start" }}
{if {= 0 {node-child-form }}
{begin
  {erase-buffer }
  {insert-string "      Form-use-#[" }
  {set-dot 17 } }
{begin
  {erase-buffer }
  {insert-string symbol.aname }
  {insert-string "      Form-use-#" }
  {insert-string {int-to-string {node-child-form} }}
  {insert-string "]" }
  {set-dot {buffer-size} }
  {setq symbol.class symbol-form.class }
  {setq symbol.start symbol-form.start } } }
```

symbol.visit

```
{setq symbol.aname symbol-name.aname }
```

symbol.enter

```
{setq symbol.node-number {node-number} }
{setq symbol.class symbol-form.class }
```

```
{setq symbol.start symbol-form.start }
```

```
symbol.delete
```

```
{setq symbols { delete-item-from-list symbols { node-number } } }
{ if { = 2 symbol-form.class }
  {setq ter-symbols { delete-item-from-list ter-symbols
                     { node-number } } } }
```

```
symbol-name.update
```

```
{ char sstring }
{ char bname }
{ int llen }
{ int count }
{ int anode }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq symbol.aname sstring }
{setq llen { size-of-list symbols } }
{ if { != 0 llen }
{ begin
  {setq count 1 }
  { while { <= count llen }
  { begin
    {setq anode { get-next-item-from-list symbols count } }
    { if { != anode symbol.node-number }
    { begin
      {setq bname { get-attr-at-node anode "aname" } }
      { if { string-compare sstring bname }
      { begin
        { message "This symbol name is already defined" }
        { quit } } } } }
      {setq count { + 1 count } } } } } }
{setq symbols { add-item-to-list symbols symbol.node-number } }
{ if { = 2 symbol.class }
  {setq ter-symbols { add-item-to-list ter-symbols
                      symbol.node-number } } }
```

```

{ if { = 1 symbol.start }
  { setq start-symbol sstring } }
{ setq symbol-name.aname sstring }

```

```

sattribute.enter
{ setq sattribute.node-number { node-number } }

```

```

sattribute-name.update
{ char sstring }
{ char bname }
{ int llen }
{ int count }
{ int anode }
{ int cnode }
{ int cclass }
{ setq cnode 0 }
{ setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
  { begin
    { message "This entry does not contain a legal name" }
    { quit } } }
{ setq llen { size-of-list attribs } }
{ if { = 0 llen }
  { begin
    { erase-buffer }
    { error-message "No attributes yet defined" } } }
{ setq count 1 }
{ while { <= count llen }
  { begin
    { setq anode { get-next-item-from-list attribs count } }
    { setq bname { get-attr-at-node anode "aname" } }
    { if { string-compare sstring bname }
      { setq cnode anode } }
    { setq count { + 1 count } } } }
{ if { = 0 cnode }
  { begin
    { erase-buffer }
    { error-message
      "No attribute by that name found in the grammar" } } }

```

```

{setq cclass { get-attr-at-node cnode "class" } }
{ if { = 3 cclass }
{ begin
  { erase-buffer }
  { error-message "This a global attribute" } } }
{setq sattribute-name.aname sstring }
{setq sattribute.aname sstring }
{setq sattribute.define 1 }
{setq sattribute.node cnode }

```

```

sdefault-value.update
{ char sstring }
{ int atype }
{setq sstring { buffer-to-string } }
{ if { = 1 sattribute.define }
{ begin
  {setq atype { get-attr-at-node sattribute.node "type" } }
  { if { & { = 1 atype } { ! { integer-test sstring } } }
  { begin
    { message "This value must be an integer" }
    { quit } } }
  {setq sattribute.value sstring } } }

```

```

saction-name.update
{ char sstring }
{ char sbstring }
{ char bname }
{ int slen }
{ int pos }
{ int sdot }
{ int llen }
{ int count }
{ int anode }
{ int cnode }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }

```



```

{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{if { = 0 pos }
{ begin
  { message "This entry does not have an extension" }
  { quit } } }
{setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{if { & { ! { string-compare sbstring ".visit" } }
  { & { ! { string-compare sbstring ".enter" } }
  { & { ! { string-compare sbstring ".update" } }
  { & { ! { string-compare sbstring ".repeat" } }
  { & { ! { string-compare sbstring ".delete" } }
  { ! { string-compare sbstring ".create" } } } } } } }
{ begin
  { message "This entry does not have a proper extension" }
  { quit } } }
{setq llen { size-of-list actions } }
{if { = 0 llen }
{ begin
  { erase-buffer }
  { error-message "No actions have been defined" } } }
{setq count 1 }
{ while { <= count llen }
{ begin
  {setq anode { get-next-item-from-list actions count } }
  {setq bname { get-attrib-at-node anode "aname" } }
  {if { string-compare sstring bname }
    {setq cnode anode } }
  {setq count { + 1 count } } } }
{if { = 0 cnode }
{ begin
  { erase-buffer }
  { error-message "No action by this name in the grammar" } } }

production-rule-form.enter
{if { = 0 { node-child-form } }
{ begin
  { erase-buffer }
  { insert-string "      Form-use-#[|]" }

```

```

    { set-dot 17 } }
{ begin
  { erase-buffer }
  { insert-string production-rule.aname }
  { insert-string "    Form-use-#" }
  { insert-string { int-to-string { node-child-form } } }
  { insert-string "]" }
  { set-dot { buffer-size } } } }

```

```

production-rule.visit
{ setq production-rule.aname production-number.aname }

```

```

production-rule.delete
{ setq products { delete-item-from-list products { node-number } } }

```

```

production-rule.enter
{ setq production-rule.node-number { node-number } }

```

```

production-number.update
{ char sstring }
{ char bname }
{ int llen }
{ int count }
{ int anode }
{ setq sstring { buffer-to-string } }
{ if { ! { integer-test sstring } } }
{ begin
  { message "This entry must be an integer" }
  { quit } } }
{ setq count { string-to-int sstring } }
{ if { = 0 count } }
{ begin
  { message "Production number of 0 is not allowed" }
  { quit } } }
{ setq llen { size-of-list products } }
{ if { != 0 llen } }
{ begin

```

```

{ setq count 1 }
{ while { <= count llen }
{ begin
  { setq anode { get-next-item-from-list products count } }
  { if { != anode production-rule.node-number }
  { begin
    { setq bname { get-attrib-at-node anode "aname" } }
    { if { string-compare sstring bname }
    { begin
      { message
        "A production by this number already defined" }
      { quit } } } } }
    { setq count { - 1 count } } } } } }
{ setq products { add-item-to-list products
  production-rule.node-number } }
{ setq production-rule.aname sstring }
{ setq production-number.aname sstring }

```

left-hand-symbol.update

```

{ char sstring }
{ char bname }
{ int anode }
{ int cnode }
{ int count }
{ int llen }
{ setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{ setq cnode 0 }
{ setq llen { size-of-list symbols } }
{ if { = 0 llen }
{ begin
  { erase-buffer }
  { error-message "No Symbols defined yet" } } }
{ setq count 1 }
{ while { <= count llen }
{ begin
  { setq anode { get-next-item-from-list symbols count } }

```

```

    { setq bname { get-attrib-at-node anode "aname" } }
    { if { string-compare sstring bname }
      { setq cnode anode } }
    { setq count { + 1 count } } } }
{ if { = 0 cnode }
{ begin
  { erase-buffer }
  { error-message "No symbol by that name in the grammar" } } }
{ setq cnode 0 }
{ setq llen { size-of-list ter-symbols } }
{ if { != 0 llen }
{ begin
  { setq count 1 }
  { while { <= count llen }
  { begin
    { setq anode { get-next-item-from-list ter-symbols count } }
    { setq bname { get-attrib-at-node anode "aname" } }
    { if { string-compare bname sstring }
      { setq cnode anode } }
    { setq count { + 1 count } } } } }
  { if { != 0 cnode }
  { begin
    { erase-buffer }
    { error-message "This symbol is a terminal symbol" } } } } }
{ setq left-hand-symbol.aname sstring }

```

handle-name.update

```

{ char sstring }
{ char bname }
{ int anode }
{ int cnode }
{ int count }
{ int llen {
{ setq sstring { buffer-to-string } }
{ setq cnode 0 }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{ setq llen { size-of-list symbols } }

```

```

{ if { = 0 llen }
{ begin
  { erase-buffer }
  { error-message "No symbols defined in the grammar" } } }
{ setq count 1 }
{ while { <= count llen }
{ begin
  { setq anode { get-next-item-from-list symbols count } }
  { setq bname { get-attrib-at-node anode "aname" } }
  { if { string-compare bname sstring }
    { setq cnode anode } }
  { setq count { + 1 count } } } }
{ if { = 0 cnode }
{ begin
  { erase-buffer }
  { error-message
    "No symbol by this name defined in the grammar" } } }
{ setq handle-name.aname sstring }

```

```

kleene-star.update
{ char sstring }
{ int count }
{ setq sstring { buffer-to-string } }
{ setq count 0 }
{ if { string-compare sstring "y" }
  { setq count 1 } }
{ if { string-compare sstring "n" }
  { setq count 2 } }
{ if { string-compare sstring "" }
  { setq count 2 } }
{ if { = 0 count }
{ begin
  { message "Values of y or n or (blank) are only allowed" }
  { quit } } }
{ if { & { = 1 count } { ! { = 1 operation.type }
  { = 0 operation.type } } }
{ begin
  { setq operation.type 1 }
  { return } } }
{ if { & { = 2 count } { = 1 operation.type } }

```

```

{ begin
  {setq operation.type 0 }
  { return } } }
{ if { = 1 count }
{ begin
  { message "More than one operation defined" }
  { quit } } }

```

```

plus.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 0 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 2 } }
{ if { string-compare sstring "" }
  {setq count 2 } }
{ if { = 0 count }
{ begin
  { message "Values of y or n or (blank) are only allowed" }
  { quit } } }
{ if { & { = 1 count } { | { = 0 operation.type }
                               { = 2 operation.type } } }
{ begin
  {setq operation.type 2 }
  { return } } }
{ if { & { = 2 count } { = 2 operation.type } }
{ begin
  {setq operation.type 0 }
  { return } } }
{ if { = 1 count }
{ begin
  { message "More than one operation defined" }
  { quit } } }

```

```

zero-one.update
{ char sstring }

```

```

{ int count }
{setq sstring { buffer-to-string } }
{setq count 0 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 2 } }
{ if { string-compare sstring "" }
  {setq count 2 } }
{ if { = 0 count }
{ begin
  { message "Values of y or n or (blank) are only allowed" }
  { quit } } }
{ if { & { = 1 count } { { = 0 operation.type }
                        { = 3 operation.type } } }
{ begin
  {setq operation.type 3 }
  { return } } }
{ if { & { = 2 count } { = 3 operation.type } }
{ begin
  {setq operation.type 0 }
  { return } } }
{ if { = 1 count }
{ begin
  { message "More than one operation defined" }
  { quit } } }

```

```

heading.update
{ char sstring }
{setq sstring { buffer-to-string } }
{ if { > 50 { length sstring } }
{ begin
  { message "Heading cannot exceed 50 characters in length" }
  { quit } } }

```

```

number-of-lines.update
{ char sstring }
{setq sstring { buffer-to-string } }
{ if { ! { integer-test sstring } }

```

```

{ begin
  { message "This entry must contain an integer" }
  { quit } } }
{ if { } 2 { string-to-int sstring } }
{ begin
  { message "Must have at least 2 lines for the display" }
  { quit } } }

```

entry-updatable.update

```

{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 0 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 2 } }
{ if { string-compare sstring "" }
  {setq count 2 } }
{ if { = 0 count }
{ begin
  { message "Values of y or n or <blank> are only allowed" }
  { quit } } } }

```

root-number.update

```

{ char sstring }
{ char bname }
{ int llen }
{ int count }
{ int anode }
{ int cnode }
{setq sstring { buffer-to-string } }
{ if { ! { integer-test sstring } }
{ begin
  { message "This entry must be an integer" }
  { quit } } }
{setq count { string-to-int sstring } }
{ if { = 0 count }
{ begin

```



```

    { message "Production number of 0 is not allowed" }
    { quit } } }
{setq llen { size-of-list products } }
{ if { = 0 llen }
{ begin
    { erase-buffer }
    { error-message "No production rules have been defined" } } }
{setq count 1 }
{ while { <= count llen }
{ begin
    {setq anode { get-next-item-from-list products count } }
    {setq bname { get-attrib-at-node anode "aname" } }
    { if { string-compare bname sstring }
        {setq cnode anode } }
    {setq count { - 1 count } } } }
{ if { = 0 cnode }
{ begin
    { erase-buffer }
    { error-message
        "No production by this number defined in the grammar" } } }
{setq llen { size-of-list bforms } }
{ if { = 0 llen }
{ begin
    { erase-buffer }
    { error-message "No blankforms have been defined" } } }
{setq count 1 }
{ while { <= count llen }
{ begin
    {setq anode { get-next-item-from-list bforms count } }
    { if { != anode form-definition.node-number }
    { begin
        {setq bname { get-attrib-at-node anode "aname" } }
        { if { string-compare bname sstring }
        { begin
            { message
                "A form beginning with this production already defined" }
            { quit } } } } }
        {setq count { - 1 count } } } }
{setq form-definition.aname sstring }

```

```
form-definition.enter  
{ setq form-definition.node-number { node-number } }
```

```
form-definition.repeat  
{ setq form-definition.node-number { node-number } }
```

```
form-definition.delete  
{ setq bforms { delete-item-from-list bforms { node-number } } }
```

```

{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { string-compare sstring "" }
{ begin
  {setq eqn-output "" }
  {setq eqn-name "" }
  {setq eqn-name-updated 1 }
  {return } } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{if { != 0 pos }
{ begin
  { message "The extension will be put on by the system" }
  { quit } } }
{setq eqn-name sstring }
{setq eqn-name-updated 1 }

```

eopt1.update

```

{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { - 2 count }
{ begin
  { message "Values of y or n or ( ) are only allowed" }
  { quit } } }
{setq eqntott.truth count }

```

```

eqn-input.update
{ char sstring }
{ char sbstring }
{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{ if { = 0 pos }
{ begin
  { message "This entry does not have an extension" }
  { quit } } }
{setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{ if { ! { string-compare sbstring ".eqn" } }
{ begin
  { message "This entry does not have the .eqn extension" }
  { quit } } }
{setq eqntott-input sstring }

```

```

eqntott-input.enter
{ erase-buffer }
{ insert-string eqntott-input }

```

```

eqntott-output.visit
{ if { = 1 eqn-name-updated }
{ begin
  { erase-buffer }
  { insert-string eqn-output }
  {setq eqn-name-updated 0 } } }

```

```

eqntott-output.update
{ char sstring }

```

```

    { setq peg-options.updated 0 }
    { if { string-compare peg-output "" }
      { return } }
    { setq peg-output "" }
    { setq peg-name-updated 1 }
    { return } } }
{ setq sstring "Executing - peg " }
{ if { ! { string-compare peg-options.truth-table-name "" } }
  { setq sstring { concat sstring "-t " } } }
{ setq sstring { concat sstring peg-input } }
{ if { = 5 prod }
{ begin
  { setq peg-output { concat peg-name ".eqn" } }
  { setq eqntott-input peg-output }
  { setq peg-name-updated 1 }
  { setq sstring { concat sstring " ) " peg-output } }
  { message sstring }
  { setq peg-options.updated 0 }
  { return } } }
{ if { = 8 prod }
{ begin
  { setq peg-output { concat peg-name ".tbl" } }
  { setq peg-name-updated 1 }
  { setq tpla-input peg-output }
  { setq sstring { concat sstring " | eqntott ) " peg-output } }
  { message sstring }
  { setq peg-options.updated 0 }
  { return } } }
{ if { = 12 prod }
{ begin
  { setq peg-output { concat peg-name ".cif" } }
  { setq mextra-input peg-output }
  { setq peg-name-updated 1 }
  { setq sstring { concat sstring
    " | eqntott | tpla -c -s Bcis -I -O -o "
    peg-output } }
  { message sstring }
  { setq peg-options.updated 0 }
  { return } } }

```

```

{ begin
  { setq peg-output "" }
  { setq peg-name "" }
  { setq peg-name-updated 1 }
  { return } } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{ setq slen { length sstring } }
{ setq sdot { string-to-char "." } }
{ setq pos { find-char-in-string sstring sdot } }
{ if { != 0 pos }
{ begin
  { message "The extension will be put on by the system" }
  { quit } } }
{ setq peg-name sstring }
{ setq peg-name-updated 1 }

```

```

peg-input.visit
{ setq pegs.updated 0 }

```

```

program.visit
{ setq peg.updated 0 }

```

```

peg-options.enter
{ setq peg-options.updated { get-attr-at-node { node-parent }
                             "updated" } }

```

```

output.visit
{ char sstring }
{ int prod }
{ if { = 0 peg-options.updated }
  { return } }
{ setq prod { node-production } }
{ if { = 0 prod }
{ begin

```

Finally, we present the set of procedural components that was implemented. The first part of each name indicates the symbol to which the procedural component is tied to. The last part of each name indicates under which condition this procedural component will be fired.

```
tt-filename.update
{ char sstring }
{ setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
  { begin
    { message "This entry does not contain a legal name" }
    { quit } } }
{ setq peg-options.truth-table-name sstring }
```

```
peg-input.update
{ setq peg-input { buffer-to-string } }
{ setq pegs.updated 1 }
```

```
program.update
{ setq peg-input "" }
{ setq peg.updated 1 }
```

```
output-file.visit
{ if { = 1 peg-name-updated }
  { begin
    { erase-buffer }
    { insert-string peg-output }
    { setq peg-name-updated 0 } } }
```

```
output-file.update
{ char sstring }
{ int slen }
{ int sdot }
{ int pos }
{ setq sstring { buffer-to-string } }
{ if { string-compare sstring "" }
```

Name	Class	Type
truth	Synth	int
truth-table-name	Synth	str
peg-input	Global	str
peg-name-updated	Global	int
peg-output	Global	str
peg-name	Global	str
updated	Inherit	int
eqntott-input	Global	str
eqn-name	Global	str
eqn-output	Global	str
eqn-name-updated	Global	int
synch	Synth	int
express	Synth	int
reduce	Synth	int
redundant	Synth	int
human	Synth	int
ninputs	Synth	int
products	Synth	int
noutputs	Synth	int
tpla-input	Global	str
tpla-name	Global	str
tpla-output	Global	str
tpla-name-updated	Global	int
ground	Synth	int
stretch	Synth	int
verbose	Synth	int
style	Synth	str
lambda	Synth	int
template	Synth	str
mextra-input	Global	str
mextra-output	Global	str
mextra-name	Global	str
mextra-name-updated	Global	int
scale-a	Synth	int
scale-b	Synth	int
capacitance	Synth	int
input	Synth	list
output	Synth	str
patch	Synth	str
name	Inherit	str

$\langle \text{mextra3} \rangle$
 $\langle \text{mextra4} \rangle$
 $\langle \text{esim1} \rangle$

14. $\langle \text{eqntott-options} \rangle ::= \langle \text{tpla} \rangle$
15. $\langle \text{eqntott-options} \rangle ::= \langle \text{mextra} \rangle$
16. $\langle \text{type} \rangle ::= \langle \text{mextra} \rangle$
17. $\langle \text{type} \rangle ::= \langle \text{merge} \rangle$
18. $\langle \text{merge} \rangle ::= \langle \text{minput} \rangle^+ \langle \text{moutput} \rangle \langle \text{drc} \rangle \langle \text{cif} \rangle$
19. $\langle \text{cif} \rangle ::= \langle \text{mextra} \rangle$
20. $\langle \text{communication} \rangle ::= \langle \text{merge} \rangle$
21. $\langle \text{esim1} \rangle ::= \langle \text{esim} \rangle$
22. $\langle \text{esim} \rangle ::= \langle \text{patchfile} \rangle \langle \text{input} \rangle^+ \langle \text{outputs} \rangle \langle \text{modify} \rangle$
23. $\langle \text{fsm-specification} \rangle ::= \langle \text{pegs} \rangle$
24. $\langle \text{pegs} \rangle ::= \langle \text{peg-input} \rangle \langle \text{output-file} \rangle \langle \text{peg-options} \rangle$

Next, we will show the list of attribute types that were used in the VLSI method. These attribute types are assigned to the grammar symbols.

6. $\langle \text{eqntott} \rangle ::= \langle \text{eqntott-input} \rangle$
 $\langle \text{eqntott-output} \rangle$
 $\langle \text{eopt1} \rangle$
 $\langle \text{eopt2} \rangle$
 $\langle \text{eopt3} \rangle$
 $\langle \text{eopt4} \rangle$
 $\langle \text{eopt5} \rangle$
 $\langle \text{eopt6} \rangle$
 $\langle \text{eopt7} \rangle$
 $\langle \text{eopt8} \rangle$
 $\langle \text{eopt9} \rangle$
 $\langle \text{eqntott-options} \rangle$
7. $\langle \text{eqntott1} \rangle ::= \langle \text{eqntott} \rangle$
8. $\langle \text{output} \rangle ::= \langle \text{tpla} \rangle$
9. $\langle \text{tpla} \rangle ::= \langle \text{tpla-input} \rangle$
 $\langle \text{tpla-output} \rangle$
10. $\langle \text{tpla-output} \rangle ::= \langle \text{filename} \rangle$
 $\langle \text{tpla2} \rangle$
 $\langle \text{tpla3} \rangle$
 $\langle \text{tpla4} \rangle$
 $\langle \text{tpla5} \rangle$
 $\langle \text{tpla6} \rangle$
 $\langle \text{tpla7} \rangle$
 $\langle \text{tpla8a} \rangle$
 $\langle \text{tpla8b} \rangle$
 $\langle \text{tpla9} \rangle$
 $\langle \text{tpla0} \rangle$
 $\langle \text{type} \rangle$
11. $\langle \text{tpla1} \rangle ::= \langle \text{tpla} \rangle$
12. $\langle \text{output} \rangle ::= \langle \text{mextra} \rangle$
13. $\langle \text{mextra} \rangle ::= \langle \text{mextra-input} \rangle$
 $\langle \text{mextra-output} \rangle$
 $\langle \text{mextra1} \rangle$
 $\langle \text{mextra2} \rangle$

VLSI-FORM-8

PEG With Input File

Form-use-# []

PEG Input Filename:

Output Filename:

Options:

Truth table filename:

{3,4,5} Output:

3. Generate equations for EQNTOTT
4. Compile PEG and pipe through EQNTOTT with standard options
5. Compile PEG and pipe through EQNTOTT and TPLA with standard options

Form-use-# []

Next, the list of all the grammar production rules are presented. * represents the kleene star; and + is the plus function. The production rules are presented using grammars with right regular parts.

1. $\langle \text{overall-specification} \rangle ::= \langle \text{fsm-specification} \rangle$
 $\langle \text{eqn-input} \rangle$
 $\langle \text{eqntott1} \rangle$
 $\langle \text{tpla-input-file} \rangle$
 $\langle \text{tpla1} \rangle$
 $\langle \text{communication} \rangle$
2. $\langle \text{fsm-specification} \rangle ::= \langle \text{peg} \rangle$
3. $\langle \text{peg} \rangle ::= \langle \text{program} \rangle$
 $\langle \text{output-file} \rangle$
 $\langle \text{peg-options} \rangle$
4. $\langle \text{peg-options} \rangle ::= \langle \text{truth-table-filename} \rangle$
 $\langle \text{output} \rangle$
5. $\langle \text{output} \rangle ::= \langle \text{eqntott} \rangle$

VLSI-FORM-5**MEXTRA**

Form-use-#[]

Input File:

Output Filename:

Scale Numerator:

Scale Denominator:

Supress Calculation of Capacitance:

Append Unique Suffixes to Names:

{7} ESIM: Form-use-#[]

VLSI-FORM-6**MERGE**

Form-use-#[]

Input Filename [more?]:

Output Filename:

DRC:

{5} CIF: Form-use-#[]

VLSI-FORM-7**ESIM**

Form-use-#[]

Patchfile name:

Input Filename [more?]:

Output Filename:

Modify:

VLSI-FORM-4

TPLA

Form-use-#[]

Input File:

Options:

Output File:

Clock the Inputs to the PLA:

Clock the Outputs to the PLA:

Insert Extra Ground at n Rows and Columns:

Stretch Power and Ground by n Lambda:

Show How PLA was Constructed:

Show what TPLA is Doing:

Styles of PLAs Available:

1. Buried contacts, nMOS, cis version
2. Buried contacts, nMOS, trans version
3. Mead Conway design rules, butting contacts, nMOS, cis version
4. Mead Conway design rules, butting contacts, nMOS, trans version
5. Buried contacts, nMOS, cis version, protection frames and terminals
6. Buried contacts, nMOS, trans version, protection frames and terminals

Select Style of PLA:

Lambda in Centimicrons:

Name of Template to use:

{5,6} Output Format:

5. Generate PLA for TPLA and go to MEXTRA
6. Generate PLA for TPLA and go to CEASAR for further refinement

Form-use-#[]

VLSI-FORM-3

EQNTOTT

Form-use-#[]

Input Filename:

Output Filename:

Output Truth Table:

Allow Input Names To Be Same As Output:

Output Variables May be Used in Expressions:

Reduce Truth Table:

No Redundant Minterms:

Human Readable Truth Table:

Output Number of Inputs:

Output Number of Product Terms:

Output Number of Outputs:

{4,5} Options:

4. Generate truth tables for TPLA

5. Compile EQNTOTT and pipe through TPLA with standard options

Form-use-#[]

VLSI-FORM-2

PEG

Form-use-# {}

Program:

Output Filename:

Options:

Truth table filename:

{3,4,5} Output:

3. Generate equations for EQNTOTT
4. Compile PEG and pipe through EQNTOTT with standard options
5. Compile PEG and pipe through EQNTOTT and TPLA with standard options

Form-use-# {}

Appendix B

Implementation of a VLSI Method

In this appendix, the blankforms, the attributed grammar production rules, and the procedural components for the VLSI method are presented. This is a small method with 8 blankforms, 24 production rules, 40 attribute types, and 56 procedural components. First, we will show the blankforms.

VLSI-FORM-1	Overall Specification	Form-use-#[]
--------------------	------------------------------	--------------

{2,8} Finite State Machine Specification:

- 2. PEG input via the programmer
- 8. PEG input via an input file

Form-use-#[]

EQNTOTT Input Filename:

{3} EQNTOTT: Form-use-#[]

TPLA Input Filename:

{4} TPLA: Form-use-#[]

{6} Communication: Form-use-#[]

```
{setq eqntott.updated 1 }
```

```
eopt2.update
```

```
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{setq eqntott.synch count }
{setq eqntott.updated 1 }
```

```
eopt3.update
```

```
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{setq eqntott.express count }
{setq eqntott.updated 1 }
```

```

eopt4.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{setq eqntott.reduce count }
{setq eqntott.updated 1 }

```

```

eopt5.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{setq eqntott.redundant count }
{setq eqntott.updated 1 }

```

```

eopt6.update
{ char sstring }
{ int count }

```

```

{ setq sstring { buffer-to-string } }
{ setq count 2 }
{ if { string-compare sstring "y" }
  { setq count 1 } }
{ if { string-compare sstring "n" }
  { setq count 0 } }
{ if { string-compare sstring "" }
  { setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{ setq eqntott.human count }
{ setq eqntott.updated 1 }

```

```

eopt7.update
{ char sstring }
{ int count }
{ setq sstring { buffer-to-string } }
{ setq count 2 }
{ if { string-compare sstring "y" }
  { setq count 1 } }
{ if { string-compare sstring "n" }
  { setq count 0 } }
{ if { string-compare sstring "" }
  { setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{ setq eqntott.ninputs count }
{ setq eqntott.updated 1 }

```

```

eopt8.update
{ char sstring }
{ int count }
{ setq sstring { buffer-to-string } }
{ setq count 2 }
{ if { string-compare sstring "y" }

```

```

    { setq count 1 } }
  { if { string-compare sstring "n" }
    { setq count 0 } }
  { if { string-compare sstring "" }
    { setq count 0 } }
  { if { = 2 count }
  { begin
    { message "Values of <y> or <n> or < > are only allowed" }
    { quit } } }
  { setq eqntott.products count }
  { setq eqntott.updated 1 }

```

```

eopt9.update
{ char sstring }
{ int count }
{ setq sstring { buffer-to-string } }
{ setq count 2 }
{ if { string-compare sstring "y" }
  { setq count 1 } }
{ if { string-compare sstring "n" }
  { setq count 0 } }
{ if { string-compare sstring "" }
  { setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{ setq eqntott.noutputs count }
{ setq eqntott.updated 1 }

```

```

eqntott-options.visit
{ char sstring }
{ int prod }
{ if { = 0 eqntott.updated }
  { return } }
{ setq prod { node-production } }
{ if { = 0 prod }
{ begin
  { setq eqntott.updated 0 }

```

```

    { if { string-compare eqn-output "" }
      { return } }
    { setq eqn-output "" }
    { setq eqn-name-updated 1 }
    { return } } }
  { setq sstring "Executing - eqntott " }
  { if { = 1 eqntott.truth }
    { setq sstring { concat sstring "-l " } } }
  { if { = 1 eqntott.synch }
    { setq sstring { concat sstring "-f " } } }
  { if { = 1 eqntott.express }
    { setq sstring { concat sstring "-s " } } }
  { if { = 1 eqntott.reduce }
    { setq sstring { concat sstring "-r " } } }
  { if { = 1 eqntott.redundant }
    { setq sstring { concat sstring "-R " } } }
  { if { = 1 eqntott.human }
    { setq sstring { concat sstring "-h " } } }
  { if { = 1 eqntott.ninputs }
    { setq sstring { concat sstring "-i " } } }
  { if { = 1 eqntott.products }
    { setq sstring { concat sstring "-p " } } }
  { if { = 1 eqntott.noutputs }
    { setq sstring { concat sstring "-o " } } }
  { setq sstring { concat sstring eqntott-input } }
  { if { = 14 prod }
    { begin
      { setq eqn-output { concat eqn-name ".tbl" } }
      { setq tpla-input eqn-output }
      { setq eqn-name-updated 1 }
      { setq sstring { concat sstring " ) " eqn-output } }
      { message sstring }
      { setq eqntott.updated 0 }
      { return } } }
  { if { = 15 prod }
    { begin
      { setq eqn-output { concat eqn-name ".cif" } }
      { setq mextra-input eqn-output }
      { setq eqn-name-updated 1 }
      { setq sstring { concat sstring " | tpla -c -s Bcis -I -O -o "
        eqn-output } }
    }
  }

```

```

{ message sstring }
{ setq eqntott.updated 0 }
{ return } } }

```

```

tpla-input-file.update
{ char sstring }
{ char sbstring }
{ int slen }
{ int sdot }
{ int pos }
{ setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{ setq slen { length sstring } }
{ setq sdot { string-to-char "." } }
{ setq pos { find-char-in-string sstring sdot } }
{ if { = 0 pos }
{ begin
  { message "This entry does not have an extension" }
  { quit } } }
{ setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{ if { ! { string-compare sbstring ".eqn" } }
{ begin
  { message "This entry does not have the .tbl extension" }
  { quit } } }
{ setq tpla-input sstring }

```

```

tpla-input.enter
{ erase-buffer }
{ insert-string tpla-input }

```

```

filename.visit
{ if { = 1 tpla-name-updated }
{ begin
  { erase-buffer }
  { insert-string tpla-output }

```

```
{setq tpla-name-updated 0 } } }
```

```
filename.update
{ char sstring }
{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { string-compare sstring "" }
{ begin
  {setq tpla-output "" }
  {setq tpla-name "" }
  {setq tpla-name-updated 1 }
  {return } } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{if { != 0 pos }
{ begin
  { message "The extension will be put on by the system" }
  { quit } } }
{setq tpla-name sstring }
{setq tpla-name-updated 1 }
```

```
type.visit
{ char sstring }
{ int prod }
{ if { = 0 tpla-output.updated }
  { return } }
{setq prod { node-production } }
{ if { = 0 prod }
{ begin
  {setq tpla-output.updated 0 }
  { if { string-compare tpla-output "" }
    { return } } }
```

```

    { setq tpla-output "" }
    { setq tpla-name-updated 1 }
    { return } } }
{ setq sstring "Executing - tpla " }
{ if { = 1 tpla-output.ninputs }
  { setq sstring { concat sstring "-I " } } }
{ if { = 1 tpla-output.noutputs }
  { setq sstring { concat sstring "-O " } } }
{ if { != 0 tpla-output.ground }
  { setq sstring { concat sstring "-G "
    { int-to-string tpla-output.ground } " " } } }
{ if { != 0 tpla-output.stretch }
  { setq sstring { concat sstring "-S "
    { int-to-string tpla-output.stretch } " " } } }
{ if { = 1 tpla-output.verbose }
  { setq sstring { concat sstring "-v " } } }
{ if { = 1 tpla-output.express }
  { setq sstring { concat sstring "-V " } } }
{ if { ! { string-compare tpla-output.style "" } }
  { setq sstring { concat sstring "-s "
    tpla-output.style " " } } }
{ if { != 0 tpla-output.lambda }
  { setq sstring { concat sstring "-l "
    { int-to-string tpla-output.lambda } " " } } }
{ if { ! { string-compare tpla-output.template "" } }
  { setq sstring { concat sstring "-t "
    tpla-output.template " " } } }
{ setq sstring { concat sstring tpla-input } }
{ if { = 16 prod }
{ begin
  { setq tpla-output { concat tpla-name ".cif" } }
  { setq mextra-input tpla-output }
  { setq tpla-name-updated 1 }
  { setq sstring { concat sstring "-c " tpla-name } }
  { message sstring }
  { setq tpla-output.updated 0 }
  { return } } }
{ if { = 17 prod }
{ begin
  { setq tpla-output { concat tpla-name ".ca" } }
  { setq tpla-name-updated 1 }

```



```

{setq sstring { concat sstring " -a " tpla-name } }
{ message sstring }
{setq tpla-output.updated 0 }
{ return } } }

```

```

tpla2.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{setq tpla-output.ninputs count }
{setq tpla-output.updated 1 }

```

```

tpla3.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{setq tpla-output.noutputs count }

```

```
{setq tpla-output.updated 1 }
```

```
tpla4.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{ if { ! { integer-test sstring } }
{ begin
  { message "This entry must contain an integer" }
  { quit } } }
{setq count { string-to-int sstring } }
{ if { = 0 count }
{ begin
  { message "Value of 0 is not meaningful" }
  { quit } } }
{setq tpla-output.ground count }
{setq tpla-output.updated 1 }
```

```
tpla5.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{ if { ! { integer-test sstring } }
{ begin
  { message "This entry must contain an integer" }
  { quit } } }
{setq count { string-to-int sstring } }
{ if { = 0 count }
{ begin
  { message "Value of 0 is not meaningful" }
  { quit } } }
{setq tpla-output.stretch count }
{setq tpla-output.updated 1 }
```

```
tpla6.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
```

```

{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{setq tpla-output.verbose count }
{setq tpla-output.updated 1 }

```

```

tpla7.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{setq tpla-output.express count }
{setq tpla-output.updated 1 }

```

```

tpla8b.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{ if { ! { integer-test sstring } }
{ begin
  { message "This entry must contain an integer" }

```

```

    { quit } } }
{setq count { string-to-int sstring } }
{ if { { = 0 count } { } count 6 } }
{ begin
  { message "Value not in range" }
  { quit } } }
{setq tpla-output.updated 1 }
{ if { = 1 count }
  {setq tpla-output.style "Bcis" } }
{ if { = 2 count }
  {setq tpla-output.style "Btrans" } }
{ if { = 3 count }
  {setq tpla-output.style "Mcis" } }
{ if { = 4 count }
  {setq tpla-output.style "Mtrans" } }
{ if { = 5 count }
  {setq tpla-output.style "Tcis" } }
{ if { = 6 count }
  {setq tpla-output.style "Ttrans" } }

```

```

tpla9.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{ if { ! { integer-test sstring } }
{ begin
  { message "This entry must contain an integer" }
  { quit } } }
{setq count { string-to-int sstring } }
{ if { = 0 count }
{ begin
  { message "Value of 0 is not meaningful" }
  { quit } } }
{setq tpla-output.lambda count }
{setq tpla-output.updated 1 }

```

```

tpla0.update
{ char sstring }
{setq sstring { buffer-to-string } }

```

```

{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq tpla-output.template sstring }
{setq tpla-output.updated 1 }

```

```

mextra-input.enter
{ erase-buffer }
{ insert-string mextra-input }

```

```

mextra-output.update
{ char sstring }
{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { string-compare sstring "" }
{ begin
  {setq mextra-output "" }
  {setq mextra-name "" }
  {setq mextra-name-updated 1 }
  { return } } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{if { != 0 pos }
{ begin
  { message "The extension will be put on by the system" }
  { quit } } }
{setq mextra-name sstring }
{setq mextra-name-updated 1 }

```

```

mextral.update

```

```

{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{ if { ! { integer-test sstring } }
{ begin
  { message "This entry must contain an integer" }
  { quit } } }
{setq count { string-to-int sstring } }
{ if { = 0 count }
{ begin
  { message "Value of 0 is not meaningful" }
  { quit } } }
{setq mextra.scale-a count }
{setq mextra.updated 1 }

```

```

mextra2.update
{ char sstring }
{ int count }
{ if { = 0 mextra.scale-a }
{ begin
  { erase-buffer }
  { message
    "Scale Denominator cannot be set with out Scale Numerator" }
  { return } } }
{setq sstring { buffer-to-string } }
{ if { ! { integer-test sstring } }
{ begin
  { message "This entry must contain an integer" }
  { quit } } }
{setq count { string-to-int sstring } }
{ if { = 0 count }
{ begin
  { message "Value of 0 is not meaningful" }
  { quit } } }
{setq mextra.scale-b count }
{setq mextra.updated 1 }

```

```

mextra3.update
{ char sstring }

```

VIMETH-FORM-13 VI Function Specification Form-use-#[]

Function:

VI Command Name:

Common?:

Argument [more?]:

Argument Name:

Type:I-integer,R-real,C-character,S-character string,B-boolean

Valid Value or Range [more?]:

Range:yes-y, no-n

Value:if range, lowerbound..upperbound

Definition:

Description:

VIMETH-FORM-11**How to Provide****Form-use-# []**

Object:

Operation:

Domain member:

How provided:

1-mapping

2-new operation

3-saved state

4-new object

5-not possible

Response:

VIMETH-FORM-12 **Included Object/Operation** **Form-use-# []**

Object/Operation:

Yes-y / No-n ?:

VIMETH-FORM-7 Effort to Provide Form-use-# []

{11} Providing Operation [more?]: Form-use-# []

{23} Summary of How Operations Provided: Form-use-# []

VIMETH-FORM-8 Objects/Operations to be in VI Form-use-# []

{12} Object/Operation [more?]: Form-use-# []

{15} VI Summary: Form-use-# []

VIMETH-FORM-9 Functional Specs of VI Funcs Form-use-# []

{13} VI Function [more?]: Form-use-# []

VIMETH-FORM-10 Implementation Form-use-# []

{18} Design test: Form-use-# []

{16} Generate target independent part: Form-use-# []

{17} Complete Target Dependent Modules: Form-use-# []

{20} Results of Test: Form-use-# []

{21} Prepare Documentation: Form-use-# []

VIMETH-FORM-5 Common Object/Operations Form-use-#[]

object / operations:

VIMETH-FORM-6 Noncommon Object/Operations Form-use-#[]

member / object / operations:

VIMETH-FORM-2**Identify Domain**

Form-use-# []

Name of Domain:

{3} Domain member [more?]: Form-use-# []

VIMETH-FORM-3**Domain Member Functions**

Form-use-# []

Member name:

{22} Function Set: Form-use-# []

{4} Object/Operation Set [more?]: Form-use-# []

VIMETH-FORM-4**Define Object/Operations**

Form-use-# []

Member Name:

Object name:

Operation name [more?]:

Appendix C

Implementation of the Virtual Interface Method

In this appendix, the blankforms, the attributed grammar production rules, and the procedural components for the virtual interface method are presented. This is a medium size method with 22 blankforms, 47 production rules, 15 attribute types, and 19 procedural components. First, we will show the blankforms.

VIMETH-FORM-1 Creation of a Virtual Interface Form-use-#[]

{2} Identify Domain: Form-use-#[]

{5} Identify Common Object/Operations: Form-use-#[]

{6} Identify Noncommon Object/Operations: Form-use-#[]

{7} Evaluate Effort: Form-use-#[]

{8} Identify VI Objects/Operations: Form-use-#[]

{19} Select Specification Technique: Form-use-#[]

{9} Specify Virtual Interface: Form-use-#[]

{10} Implement Virtual Interface: Form-use-#[]

```
{ message ".ca files have been changed to .cif files" }
```

```
tpla8a.enter  
{ set-dot 1 }
```

```

    { message "This entry does not have an extension" }
    { quit } } }
{setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{ if { ! { string-compare sbstring ".ca" } } }
{ begin
  { message "This entry does not have the .ca extension" }
  { quit } } }

```

```

moutput.update
{ char sstring }
{ char sbstring }
{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{ if { = 0 pos } }
{ begin
  { message "This entry does not have an extension" }
  { quit } } }
{setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{ if { ! { string-compare sbstring ".cif" } } }
{ begin
  { message "This entry does not have the .cif extension" }
  { quit } } }
{setq merge.name sstring }

```

```

drc.enter
{ message "Executing - lyra " }

```

```

cif.visit
{setq mextra-input merge.name }

```

```

minput.update
{ char sstring }
{ char sbstring }
{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{ if { = 0 pos }
{ begin
  { message "This entry does not have an extension" }
  { quit } } }
{setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{ if { ! { string-compare sbstring ".ca" } }
{ begin
  { message "This entry does not have the .ca extension" }
  { quit } } } }

```

```

minput.repeat
{ char sstring }
{ char sbstring }
{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{ if { = 0 pos }
{ begin

```

```

{ begin
  { message "This entry does not have an extension" }
  { quit } } }
{setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{ if { ! { string-compare sbstring ".out" } } }
{ begin
  { message "This entry does not have the .out extension" }
  { quit } } }
{setq esim.output sstring }
{setq esim.updated 1 }

```

```

modify.enter
{ char sstring }
{ int count }
{ int len }
{ int anode }
{ char bname }
{ if { = 0 esim.updated }
  { return } }
{setq sstring "Executing - esim " }
{ if { ! { string-compare esim.patch "" } }
  {setq sstring { concat sstring esim.patch " " } } }
{setq len { size-of-list esim.input } }
{setq count 1 }
{ while { <= count len }
{ begin
  {setq anode { get-next-item-from-list esim.input count } }
  {setq bname { get-attr-at-node anode "name" } }
  { if { ! { string-compare bname "" } }
    {setq sstring { concat sstring bname " " } } }
  {setq count { + 1 count } } } }
{ if { ! { string-compare esim.output "" } }
  {setq sstring { concat sstring "-" esim.output " " } } }
{ message sstring }
{setq esim.updated 0 }

```

```

input.delete
{setq esim.input { delete-item-from-list esim.input { node-number } } }

```



```

{ char sbstring }
{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{ if { = 0 pos }
{ begin
  { message "This entry does not have an extension" }
  { quit } } }
{setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{ if { ! { string-compare sbstring ".sim" } }
{ begin
  { message "This entry does not have the .sim extension" }
  { quit } } }
{setq input.name sstring }
{setq esim.input { add-item-to-list esim.input { node-number } } }
{setq esim.updated 1 }

```

```

outputs.update
{ char sstring }
{ char sbstring }
{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{ if { = 0 pos }

```

```

    { message "This entry does not have an extension" }
    { quit } } }
{setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{ if { ! { string-compare sbstring ".sim" } } }
{ begin
    { message "This entry does not have the .sim extension" }
    { quit } } }
{setq esim.patch sstring }
{setq esim.updated 1 }

```

```

input.update
{ char sstring }
{ char sbstring }
{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } } }
{ begin
    { message "This entry does not contain a legal name" }
    { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{ if { = 0 pos } }
{ begin
    { message "This entry does not have an extension" }
    { quit } } }
{setq sbstring { substr sstring pos { + 1 { - slen pos } } } }
{ if { ! { string-compare sbstring ".sim" } } }
{ begin
    { message "This entry does not have the .sim extension" }
    { quit } } }
{setq input.name sstring }
{setq esim.input { add-item-to-list esim.input { node-number } } }
{setq esim.updated 1 }

```

```

input.repeat
{ char sstring }

```

```

{setq prod { node-production } }
{ if { = 0 prod }
{ begin
  {setq mextra.updated 0 }
  { if { string-compare mextra-output "" }
    { return } }
  {setq mextra-output "" }
  {setq mextra-name-updated 1 }
  { return } } }
{setq sstring "Executing - mextra " }
{ if { != 0 mextra.scale-a }
  {setq sstring { concat sstring "-u "
    { int-to-string mextra.scale-a } "/"
    { int-to-string mextra.scale-b } " " } } }
{ if { = 1 mextra.capacitance }
  {setq sstring { concat sstring "-o " } } }
{ if { = 1 mextra.synch }
  {setq sstring { concat sstring "-g " } } }
{setq sstring { concat sstring mextra-name } }
{setq mextra-output { concat mextra-name ".sim" } }
{setq mextra-name-updated 1 }
{ message sstring }
{setq mextra.updated 0 }

```

```

patchfile.update
{ char sstring }
{ char sbstring }
{ int slen }
{ int sdot }
{ int pos }
{setq sstring { buffer-to-string } }
{ if { ! { name-test sstring } }
{ begin
  { message "This entry does not contain a legal name" }
  { quit } } }
{setq slen { length sstring } }
{setq sdot { string-to-char "." } }
{setq pos { find-char-in-string sstring sdot } }
{ if { = 0 pos }
{ begin

```

```

{ int count }
{setq sstring { buffer-to-string } }
{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{setq mextra.capacitance count }
{setq mextra.updated 1 }

```

```

mextra4.update
{ char sstring }
{ int count }
{setq sstring { buffer-to-string } }
{setq count 2 }
{ if { string-compare sstring "y" }
  {setq count 1 } }
{ if { string-compare sstring "n" }
  {setq count 0 } }
{ if { string-compare sstring "" }
  {setq count 0 } }
{ if { = 2 count }
{ begin
  { message "Values of <y> or <n> or < > are only allowed" }
  { quit } } }
{setq mextra.synch count }
{setq mextra.updated 1 }

```

```

esim1.visit
{ char sstring }
{ int prod }
{ if { = 0 mextra.updated }
  { return } }

```

VIMETH-FORM-15 Virtual Interface Functions Form-use-# {}

Virtual Interface:

Object/Operation Sets:

Completeness Properties:

VIMETH-FORM-16 Target Independent Part

Form-use-# []

Target Independent Skeleton:

Target Independent Code:

VIMETH-FORM-17 Target Dependent Modules

Form-use-# []

Target Dependent Skeleton:

Target Dependent Code:

VIMETH-FORM-18**Test Description**

Form-use-#[]

Test Description:

VIMETH-FORM-19**Specification Technique**

Form-use-#[]

Select Specification Technique: Choose Functional Specification to continue.

VIMETH-FORM-20**Test Results**

Form-use-#[]

Test Results:

VIMETH-FORM-21**Documentation****Form-use-#** []**Documentation:**

VIMETH-FORM-22**Function Set****Form-use-#** []**Member Name:**

Function [more?]:

VIMETH-FORM-23 Method of Providing Funcs Form-use-# []**Method of Providing:**

Next, the list of all the production rules are presented.

1. $\langle vi \rangle ::= \langle identify-domain \rangle$
 $\quad \langle identify-common \rangle$
 $\quad \langle identify-noncommon \rangle$
 $\quad \langle evaluate-effort \rangle$
 $\quad \langle identify-vi-ops \rangle$
 $\quad \langle select-spec-tech \rangle$
 $\quad \langle specify-interface \rangle$
 $\quad \langle implementation \rangle$
92. $\langle operation-set \rangle ::= \langle objects-operations \rangle$
101. $\langle identify-domain \rangle ::= \langle domain \rangle$
102. $\langle domain \rangle ::= \langle domain-name \rangle$
 $\quad \langle domain-member \rangle^*$
201. $\langle domain-member \rangle ::= \langle metafunction \rangle$
202. $\langle metafunction \rangle ::= \langle member-name \rangle$
 $\quad \langle function-set \rangle$
 $\quad \langle operation-set \rangle^*$
203. $\langle function-set \rangle ::= \langle func \rangle$
213. $\langle func \rangle ::= \langle mem-name \rangle$
 $\quad \langle function \rangle^+$
301. $\langle objects-operations \rangle ::= \langle mem-name \rangle$
 $\quad \langle object \rangle$
 $\quad \langle operation \rangle^+$
401. $\langle common \rangle ::= \langle common-list \rangle$
402. $\langle identify-common \rangle ::= \langle common \rangle$
404. $\langle summary \rangle ::= \langle prov \rangle$
414. $\langle prov \rangle ::= \langle provided-chart \rangle$
501. $\langle noncommon \rangle ::= \langle noncommon-list \rangle$

- 502. $\langle \text{identify-noncommon} \rangle ::= \langle \text{noncommon} \rangle$
- 601. $\langle \text{evaluate} \rangle ::= \langle \text{effort} \rangle^+ \langle \text{summary} \rangle$
- 602. $\langle \text{evaluate-effort} \rangle ::= \langle \text{evaluate} \rangle$
- 603. $\langle \text{effort} \rangle ::= \langle \text{eval-effort} \rangle$
- 613. $\langle \text{eval-effort} \rangle ::= \langle \text{object} \rangle$
 $\quad \langle \text{operation} \rangle$
 $\quad \langle \text{member-name} \rangle$
 $\quad \langle \text{how-provided} \rangle$
 $\quad \langle \text{response} \rangle$
- 701. $\langle \text{identify} \rangle ::= \langle \text{include-set} \rangle^* \langle \text{completeness} \rangle$
- 702. $\langle \text{identify-vi-ops} \rangle ::= \langle \text{identify} \rangle$
- 703. $\langle \text{include-set} \rangle ::= \langle \text{inc-set} \rangle$
- 704. $\langle \text{completeness} \rangle ::= \langle \text{metacompleteness} \rangle$
- 705. $\langle \text{yes-set} \rangle ::= \langle \text{vi-include-chart} \rangle$
- 706. $\langle \text{comp-prop} \rangle ::= \langle \text{complete-chart} \rangle$
- 713. $\langle \text{inc-set} \rangle ::= \langle \text{object-operation} \rangle \langle \text{include} \rangle$
- 714. $\langle \text{metacompleteness} \rangle ::= \langle \text{yes-set} \rangle \langle \text{comp-prop} \rangle$
- 801. $\langle \text{spec-tech} \rangle ::= \langle \text{technique} \rangle$
- 802. $\langle \text{select-spec-tech} \rangle ::= \langle \text{spec-tech} \rangle$
- 901. $\langle \text{specify} \rangle ::= \langle \text{vi-function} \rangle^+$
- 902. $\langle \text{specify-interface} \rangle ::= \langle \text{specify} \rangle$

- 903. $\langle \text{vi-function} \rangle ::= \langle \text{vi-func} \rangle$
- 904. $\langle \text{argument} \rangle ::= \langle \text{argument-name} \rangle$
 $\quad \langle \text{type} \rangle$
 $\quad \langle \text{values} \rangle^+$
- 913. $\langle \text{vi-func} \rangle ::= \langle \text{object-operation} \rangle$
 $\quad \langle \text{vi-name} \rangle$
 $\quad \langle \text{combool} \rangle$
 $\quad \langle \text{argument} \rangle^*$
 $\quad \langle \text{description} \rangle$
- 914. $\langle \text{values} \rangle ::= \langle \text{rangebool} \rangle$
 $\quad \langle \text{value} \rangle$
 $\quad \langle \text{definition} \rangle$
- 1001. $\langle \text{metaimplement} \rangle ::= \langle \text{design-test} \rangle$
 $\quad \langle \text{generate-tar-ind} \rangle$
 $\quad \langle \text{complete-tar-dep} \rangle$
 $\quad \langle \text{perform-test} \rangle$
 $\quad \langle \text{prepare-documentation} \rangle$
- 1002. $\langle \text{implementation} \rangle ::= \langle \text{metaimplement} \rangle$
- 1003. $\langle \text{generate-tar-ind} \rangle ::= \langle \text{tar-ind} \rangle$
- 1004. $\langle \text{complete-tar-dep} \rangle ::= \langle \text{tar-dep} \rangle$
- 1005. $\langle \text{design-test} \rangle ::= \langle \text{dtest} \rangle$
- 1006. $\langle \text{perform-test} \rangle ::= \langle \text{ptest} \rangle$
- 1007. $\langle \text{prepare-documentation} \rangle ::= \langle \text{doc} \rangle$
- 1013. $\langle \text{tar-ind} \rangle ::= \langle \text{iskeleton} \rangle$
 $\quad \langle \text{icode} \rangle$
- 1014. $\langle \text{tar-dep} \rangle ::= \langle \text{dskeleton} \rangle$
 $\quad \langle \text{dcode} \rangle$
- 1015. $\langle \text{dtest} \rangle ::= \langle \text{test-description} \rangle$

1016. $\langle \text{ptest} \rangle ::= \langle \text{test-results} \rangle$

1017. $\langle \text{doc} \rangle ::= \langle \text{documents} \rangle$

Next, we show the list of attribute types that was used in the virtual interface method. These attribute types are assigned to the grammar symbols.

Name	Class	Type
domain-name	Global	str
function-name	Synth	str
object-name	Synth	str
operation-name	Synth	str
member-name	Synth	str
function-set	Synth	list
operation-set	Inherit	list
object-set	Inherit	list
node-number	Synth	int
object-sets	Global	list
common-set	Global	list
valid	Global	int
common	Synth	int
flag	Synth	int
object-length	Global	int

Finally, we present the set of procedural components that was implemented. The first part of each name indicates the symbol to which the procedural component is tied to. The last part of each name indicates under which condition this procedural component will be fired.

```
domain.update
{ setq domain-name { buffer-to-string } }
```

```
function.update
{ char sstring }
{ char bname }
{ int llen }
{ int count }
```

```

{ int anode }
{setq sstring { buffer-to-string } }
{setq function.function-name sstring }
{setq llen { size-of-list func.function-set } }
{ if { != 0 llen }
{ begin
  {setq count 1 }
  { while { <= count llen }
  { begin
    {setq anode { get-next-item-from-list func.function-set count } }
    {setq bname { get-attrib-at-node anode "function-name" } }
    { if { string-compare sstring bname }
    { begin
      { message "This function is already defined" }
      { quit } } }
    {setq count { + 1 count } } } } } } }
{setq func.function-set { add-item-to-list func.function-set
  { node-number } } }

```

```

object.update
{ char sstring }
{setq sstring { buffer-to-string } }
{setq objects-operations.object-set { add-item-to-list
  objects-operations.object-set objects-operations.node-number } }
{setq objects-operations.object-name sstring }
{setq valid 0}

```

```

operation.update
{ char sstring }
{ char bname }
{ int llen }
{ int count }
{ int anode }
{setq sstring { buffer-to-string } }
{setq llen { size-of-list objects-operations.operation-set } }
{ if { != 0 llen }
{ begin
  {setq count 1 }
  { while { <= count llen }

```

```

{ begin
  { setq anode { get-next-item-from-list
                objects-operations.operation-set count } }
  { if { != anode { node-number } }
  {begin
    { setq bname { get-attrib-at-node anode
                  "operation-name" } }
    { if { string-compare sstring bname }
    { begin
      { message "This operation has already been used" }
      { quit } } } } }
    { setq count { + 1 count } } } } } }
{ setq objects-operations.operation-set { add-item-to-list
  objects-operations.operation-set { node-number } } }
{ setq operation.operation-name sstring }
{ setq valid 0 }

```

member-name.update

```

{ setq metafunction.member-name { buffer-to-string } }

```

metafunction.visit

```

{ setq metafunction.function-set function-set.function-set }

```

function-set.enter

```

{ setq function-set.member-name metafunction.member-name }
{ if { != 0 { node-child-form } }
  { setq function-set.function-set func.function-set } }

```

func.enter

```

{ setq func.member-name function-set.member-name }

```

mem-name.enter

```

{ erase-buffer }
{ insert-string { get-attrib-at-node { node-parent } "member-name" } }

```

```
objects-operations.enter
{ setq objects-operations.member-name operation-set.member-name }
{ setq objects-operations.node-number { node-number } }
```

```
operation-set.enter
{ setq operation-set.member-name metafunction.member-name }
{ if { = 0 { node-child-form } }
  { begin
    { erase-buffer }
    { insert-string "      Form-use-#" }
    { set-dot 17 }
    { setq operation-set.object-set { create-item-list } }
    { if { = 1 operation-set.flag }
      { begin
        { setq operation-set.flag 0 }
        { setq metafunction.object-set { delete-item-from-list
          metafunction.object-set {node-number } } } } } }
  { begin
    { erase-buffer }
    { insert-string objects-operations.object-name }
    { insert-string "      Form-use-#" }
    { insert-string { int-to-string { node-child-form } } }
    { insert-string "}" }
    { set-dot { buffer-size } }
    { setq operation-set.object-set objects-operations.object-set }
    { setq operation-set.flag 1 }
    { setq metafunction.object-set { add-item-to-list
      metafunction.object-set { node-number } } } } }
```

```
operation.repeat
{ char sstring }
{ char bname }
{ int llen }
{ int count }
{ int anode }
{ setq sstring { buffer-to-string } }
{ setq llen { size-of-list objects-operations.operation-set } }
{ if { != 0 llen }
  { begin
```

```

{setq count 1 }
{ while { <= count llen }
{ begin
  {setq anode { get-next-item-from-list
                objects-operations.operation-set count } }
  { if { != anode { node-number } }
  {begin
    {setq bname { get-attrib-at-node anode
                  "operation-name" } }
    { if { string-compare sstring bname }
    { begin
      { message "This operation has already been used" }
      { quit } } } } }
    {setq count { + 1 count } } } } } }
{setq objects-operations.operation-set { add-item-to-list
    objects-operations.operation-set { node-number } } }
{setq operation.operation-name sstring }
{setq valid 0 }

```

objects-operations.delete

```

{setq objects-operations.object-set { delete-item-from-list
    objects-operations.object-set { node-number } } }
{setq valid 0 }

```

domain-member.enter

```

{ if { = 0 { node-child-form } }
{ begin
  { erase-buffer }
  { insert-string "      Form-use-#[" }
  { set-dot 17 }
  { if { = 1 domain-member.flag }
    {setq object-sets { delete-item-from-list object-sets
                        { node-number } } } }
  {setq domain-member.flag 0 }
  {setq domain-member.object-set { create-item-list } } }
{ begin
  { erase-buffer }
  { insert-string metafunction.member-name }
  { insert-string "      Form-use-#[" }

```



```

    { insert-string { int-to-string { node-child-form } } }
    { insert-string "]" }
    { set-dot { buffer-size } }
    {setq domain-member.flag 1}
    {setq domain-member.member-name metafunction.member-name }
    {setq domain-member.object-set metafunction.object-set }
    {setq object-sets { add-item-to-list object-sets
                        { node-number } } } }
{setq object-length { size-of-list object-sets } }

```

identify-domain.enter

```

{ if { = 0 { node-child-form } }
{ begin
    { erase-buffer }
    { insert-string "      Form-use-#[[]" }
    { set-dot 17 } }
{begin
    { erase-buffer }
    { insert-string domain-name }
    { insert-string "      Form-use-#[[" }
    { insert-string { int-to-string { node-child-form } } }
    { insert-string "]" }
    { set-dot { buffer-size } } } }

```

identify-common.enter

```

{ int count }
{ int anode }
{ list alist }
{ list blist }
{ int bcount }
{ int blen }
{ int objcount }
{ int objllen }
{ int comcount }
{ int comllen }
{ int comnode }
{ char comname }
{ list comlist }
{ list objlist }

```

```

{ int objnode }
{ char objname }
{ list coplist }
{ list ooplist }
{ int coplen }
{ int ooplen }
{ int copcount }
{ int oopcount }
{ char copname }
{ char oopname }
{ int temp }
{ int bnode }
{ int cnode }
{ if { = 1 valid }
  { return } }
{ if { = 0 object-length }
  { return } }
{setq anode { get-next-item-from-list object-sets 1 } }
{setq common-set { get-attrib-at-node anode "object-set" } }
{setq comllen { size-of-list common-set } }
{ if { = 0 comllen }
  { return } }
{setq count 1 }
{ while { (<= count object-length ) }
  { begin
    {setq anode { get-next-item-from-list object-sets count } }
    {setq blist { get-attrib-at-node anode "object-set" } }
    {setq blen { size-of-list blist } }
    {setq bcount 1 }
    { while { (<= bcount blen ) }
      { begin
        {setq anode { get-next-item-from-list blist bcount } }
        {setq alist { get-attrib-at-node anode "object-set" } }
        {setq anode { get-next-item-from-list alist 1 } }
        {set-attrib-at-node anode "common" 1 }
        {setq coplist { get-attrib-at-node anode "operation-set" } }
        {setq coplen { size-of-list coplist } }
        {setq copcount 1 }
        { while { (<= copcount coplen ) }
          { begin
            {setq anode { get-next-item-from-list coplist copcount } }

```

AD-A158 182

A META SYSTEM FOR GENERATING SOFTWARE ENGINEERING
ENVIRONMENTS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON
AFB OH W L MCKNIGHT 1985 AFIT/CI/NR-85-71D

4/4

UNCLASSIFIED

F/G 9/2

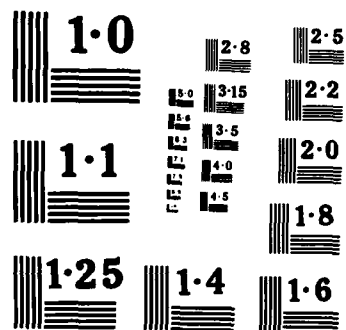
NL



END

FILED

DTIC



NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

```

        { set-attr-at-node anode "common" 1 }
        { setq copcount { + 1 copcount } } } }
    { setq bcount { + 1 bcount } } } }
    { setq count { + 1 count } } } }
{ setq count 1 }
{ while { < count object-length }
{ begin
    { setq anode { get-next-item-from-list object-sets count } }
    { setq blist { get-attr-at-node anode "object-set" } }
    { setq comllen { size-of-list blist } }
    { setq bcount { + 1 count } }
    { while { <= bcount object-length }
    { begin
        { setq anode { get-next-item-from-list object-sets bcount } }
        { setq alist { get-attr-at-node anode "object-set" } }
        { setq objllen { size-of-list alist } }
        { setq comcount 1 }
        { while { <= comcount comllen }
        { begin
            { setq anode { get-next-item-from-list blist comcount } }
            { setq comlist { get-attr-at-node anode "object-set" } }
            { setq anode { get-next-item-from-list comlist 1 } }
            { setq comname { get-attr-at-node anode
                                "object-name" } }

            { setq objcount 1 }
            { while { <= objcount objllen }
            { begin
                { setq bnode { get-next-item-from-list alist objcount } }
                { setq objlist { get-attr-at-node bnode "object-set" } }
                { setq bnode { get-next-item-from-list objlist 1 } }
                { setq objname { get-attr-at-node bnode
                                    "object-name" } }

                { if { string-compare objname comname }
                { begin
                    { setq temp { get-attr-at-node anode
                                    "common" } }
                    { setq temp { + 1 temp } }
                    { set-attr-at-node anode "common" temp }
                    { setq temp { get-attr-at-node bnode
                                    "common" } }
                    { setq temp { + 1 temp } }

```

```

{ set-attr-at-node bnode "common" temp }
{ setq anode { get-next-item-from-list comlist 1 } }
{ setq coplist { get-attr-at-node anode
                  "operation-set" } }
{ setq coplen { size-of-list coplist } }
{ setq anode { get-next-item-from-list objlist 1 } }
{ setq ooplist { get-attr-at-node anode
                  "operation-set" } }
{ setq ooplen { size-of-list ooplist } }
{ setq copcount 1 }
{ while { <= copcount coplen }
{ begin
  { setq cnode { get-next-item-from-list coplist
                  copcount } }
  { setq copname { get-attr-at-node cnode
                    "operation-name" } }
  { setq oopcount 1 }
  { while { <= oopcount ooplen }
  { begin
    { setq anode { get-next-item-from-list
                    ooplist oopcount } }
    { setq oopname { get-attr-at-node anode
                      "operation-name" } }
    { if { string-compare oopname copname }
    { begin
      { setq temp { get-attr-at-node anode
                      "common" } }
      { setq temp { + 1 temp } }
      { set-attr-at-node anode
                          "common" temp }
      { setq temp { get-attr-at-node cnode
                      "common" } }
      { setq temp { + 1 temp } }
      { set-attr-at-node cnode
                          "common" temp } } } }
      { setq oopcount { + 1 oopcount } } } } }
      { setq copcount { + 1 copcount } } } } } }
      { setq objcount { + 1 objcount } } } }
      { setq comcount { + 1 comcount } } } }
      { setq bcount { + 1 bcount } } } }
      { setq count { + 1 count } } } } }

```

```
{setq valid 1 }
```

```
common.enter
```

```
{ char comname }
{ list comlist }
{ int comlen }
{ int anode }
{ int comcount }
{ int comflag }
{ list coplist }
{ int coplen }
{ int copcount }
{ int cnode }
{ erase-buffer }
{setq comlen { size-of-list common-set } }
{ if { = 0 comlen }
{ begin
  { insert-string "There are no common objects and operations!" }
  { return } } }
{setq comcount 1 }
{ while { <= comcount comlen }
{ begin
  {setq anode { get-next-item-from-list common-set comcount } }
  {setq comlist { get-attrib-at-node anode "object-set" } }
  {setq anode { get-next-item-from-list comlist 1 } }
  {setq comflag { get-attrib-at-node anode "common" } }
  { if { = object-length comflag }
  { begin
    {setq comname { get-attrib-at-node anode "object-name" } }
    { insert-string "
    " }
    { insert-string comname }
    { insert-string "
  " }
  {setq coplist { get-attrib-at-node anode "operation-set" } }
  {setq coplen { size-of-list coplist } }
  { if { != 0 coplen }
  { begin
    {setq copcount 1 }
    { while { <= copcount coplen }
```

```

{ begin
  { setq cnode { get-next-item-from-list coplist
                copcount } }
  { setq comflag { get-attrib-at-node cnode "common" } }
  { if { = object-length comflag }
    { begin
      { setq comname { get-attrib-at-node cnode
                      "operation-name" } }
      { insert-string " " }
      { insert-string comname }
      { insert-string "
    } } }
    { setq copcount { + 1 copcount } } } } } } } } } }
  { setq comcount { + 1 comcount } } } } } } } } } }

```

operation.delete

```

{ setq objects-operations.operation-set { delete-item-from-list
  objects-operations.operation-set { node-number } } }
{ setq valid 0 }

```

noncommon.enter

```

{ char aname }
{ int count }
{ int aflag }
{ list alist }
{ char comname }
{ list comlist }
{ int comlen }
{ int anode }
{ int comcount }
{ int comflag }
{ list coplist }
{ int coplen }
{ int copcount }
{ int cnode }
{ erase-buffer }
{ if { = 0 object-length }
  { begin
    { insert-string "There are no noncommon objects and operations!" }

```


END

FILMED

10-85

DTIC